A CONSTRUCTIVE PLACEMENT BY PARTITIONING ALGORITHM USING A PERFORMANCE-ORIENTED SIXTEEN-WAY PLANAR PARTITIONING ALGORITHM

BY

JAMES PATRICK STAPLETON
B.S., University of Illinois, 1988

THESIS
Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 1989

Urbana, Illinois
ABSTRACT

This thesis introduces a new placement algorithm which is based on a sixteen-way planar constructive partitioning algorithm. The placement algorithm begins with a gate-level schematic circuit representation and schedules a series of partitioning problems that, collectively, form a complete placement solution. The algorithm is based on the concept of placement by partitioning and uses a sixteen-way partitioning algorithm whose performance is enhanced by parallel processing. The partitioning algorithm is particularly interesting because of its ability to partition into as many as sixteen partitions. In addition to considering partition cut-crossings, the partitioning algorithm also considers the relative positioning of the partitions in the plane. The performance of the partitioning algorithm is enhanced by the use of parallel processing, bit-oriented operations, custom memory allocation, and specialized data structures. This thesis also includes the results of performance experiments that were performed on the algorithm.
I would like to thank Professor David Knapp for his guidance during the course of this research and Dr. Ricardo Uribe for acting as co-advisor and for allowing me the opportunity to be a teaching assistant for EE249.

I would also like to thank my fiancee, Michelle Ohms, for her consistent love and support during the past year. I thank her for the time she spent reading and correcting drafts of this thesis. Finally, a special thanks to Bob, Dennis, Drew, Goran, Ken, Mike, Rich and Scott for being such good friends during my graduate work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
</tr>
<tr>
<td>2</td>
<td>SURVEY OF PREVIOUS WORK</td>
</tr>
<tr>
<td></td>
<td>2.1 Constructive Placement Algorithms</td>
</tr>
<tr>
<td></td>
<td>2.1.1 Cluster growth</td>
</tr>
<tr>
<td></td>
<td>2.1.2 Force directed</td>
</tr>
<tr>
<td></td>
<td>2.1.3 Branch-and-bound algorithms</td>
</tr>
<tr>
<td></td>
<td>2.1.4 Placement by partitioning</td>
</tr>
<tr>
<td></td>
<td>2.2 Iterative Placement Algorithms</td>
</tr>
<tr>
<td></td>
<td>2.2.1 Interchange</td>
</tr>
<tr>
<td></td>
<td>2.2.2 Force-directed algorithms</td>
</tr>
<tr>
<td></td>
<td>2.2.3 Steinberg's algorithm</td>
</tr>
<tr>
<td></td>
<td>2.2.4 Simulated annealing</td>
</tr>
<tr>
<td>3</td>
<td>PLACEMENT ALGORITHM</td>
</tr>
<tr>
<td></td>
<td>3.1 Partitioning Algorithm</td>
</tr>
<tr>
<td></td>
<td>3.1.1 Seed assignment</td>
</tr>
<tr>
<td></td>
<td>3.1.2 Neighborhoods and affinities</td>
</tr>
<tr>
<td></td>
<td>3.1.3 Maintaining balance</td>
</tr>
<tr>
<td></td>
<td>3.1.4 Arbitration and assignment</td>
</tr>
<tr>
<td></td>
<td>3.2 Global Awareness</td>
</tr>
<tr>
<td></td>
<td>3.2.1 Imaginary components</td>
</tr>
<tr>
<td></td>
<td>3.2.2 Considering only placed components</td>
</tr>
<tr>
<td></td>
<td>3.2.3 Considering all components</td>
</tr>
<tr>
<td></td>
<td>3.3 Scheduling of Partitioning Problems</td>
</tr>
<tr>
<td></td>
<td>3.3.1 Number and sizes of partitions</td>
</tr>
<tr>
<td></td>
<td>3.3.2 Depth-first or breadth-first</td>
</tr>
<tr>
<td></td>
<td>3.3.3 Ascending or descending partition sizes</td>
</tr>
<tr>
<td></td>
<td>3.3.4 Use of sixteen-way partitions</td>
</tr>
<tr>
<td></td>
<td>3.4 Implementation</td>
</tr>
<tr>
<td></td>
<td>3.4.1 Data structures used</td>
</tr>
<tr>
<td></td>
<td>3.4.2 Use of bit-oriented operations</td>
</tr>
<tr>
<td></td>
<td>3.4.3 Memory allocation</td>
</tr>
<tr>
<td></td>
<td>3.4.4 Parallelization</td>
</tr>
</tbody>
</table>
4 EXPERIMENTAL RESULTS ........................................................................................................ 46
  4.1 Test Cases Used .............................................................................................................. 47
  4.2 Performance Measures .................................................................................................. 48
  4.3 Effect of Other Users of Sequent Balance 8000 ......................................................... 49
  4.4 Speedup Analysis ......................................................................................................... 50
  4.5 Variation of Placement Parameters .............................................................................. 53
  4.6 Performance of Optimized Placement Algorithm ....................................................... 62
  4.7 Simulated Annealing As a Benchmark Algorithm ...................................................... 66
  4.8 Simulated Annealing Results vs. Our Results ............................................................ 67

5 FUTURE RESEARCH ............................................................................................................. 72
  5.1 Variation of Neighborhood Weighting Factors .......................................................... 72
  5.2 Aspect Ratio As User Input .......................................................................................... 73
  5.3 Preplaced Components ................................................................................................. 75
  5.4 Fractional Imaginary Components ............................................................................. 75
  5.5 Use As Input to Iterative Placement Algorithm ......................................................... 76

6 CONCLUSIONS .................................................................................................................... 77

REFERENCES ................................................................................................................................ 79
CHAPTER 1
INTRODUCTION

This thesis introduces a new placement algorithm which is based on a sixteen-way planar constructive partitioning algorithm. The placement algorithm begins with a gate-level schematic circuit representation and schedules a series of partitioning problems that, collectively, form a complete placement solution. The algorithm is based on the concept of placement by partitioning and uses a sixteen-way partitioning algorithm whose performance is enhanced by parallel processing. The partitioning algorithm is particularly interesting because of its ability to partition into as many as sixteen partitions. In addition to considering partition cut-crossings, the partitioning algorithm also considers the relative positioning of the partitions in the plane. The performance of the partitioning algorithm is enhanced by the use of parallel processing, bit-oriented operations, custom memory allocation, and special data structures.

This chapter introduces the placement problem, its associated terminology, and the objectives which it attempts to achieve. Placement is the portion of the layout process that determines where the physical components of a circuit will be placed.
The layout of a circuit consists of three interdependent phases: partitioning, placement and routing. Partitioning consists of dividing a circuit into groups so that the interconnection between the groups is minimized. This can be used, for example, to assign chips to boards in a multiple board design. The objective, in this case, would be to minimize the number of signals that must be passed through the backplane.

The second phase of the layout is placement. The placement problem, as applied in this thesis, consists of assigning components to positions on a carrier as shown in Figure 1. The components are the objects to be positioned, while the carrier is the plane on which the components are to be positioned. An example of a component and corresponding carrier would be standard cells on a chip. In this case, the components would be the standard cells, and the carrier would be the silicon substrate. The position of a component on a carrier can have different meanings and constraints, depending on the technology which is being used. For instance, if the goal were to position gates on a VLSI gate array, then the positions of the components on the carrier would be constrained to the two-dimensional array of uncommitted transistors that makes up the gate array chip. In contrast, a general cell design would allow complete flexibility in the positioning of components, but would require that the resulting placement be as compact as possible. When performing placement, the primary consideration is the connectivity of
The Schematic Circuit Representation

Figure 1: Basic Placement Problem
the components to be placed. The components' pins are connected by nets which represent signals in the circuit. A net uniformly connects two or more components of a circuit together, and is defined by the set of components that it connects.

The objective of the placement problem is to place the components on the carrier so that the routing of the nets can be accomplished with as little space as possible. These goals are achieved by positioning components that are connected by nets in close proximity with each other to keep wire lengths small, and by attempting to keep congestion under control by spreading the connections evenly across the carrier. To evaluate how well these objectives have been met, a cost function is created. The cost function may be the sum of the wire lengths, the percentage of nets that can be successfully routed, the wire buildup in the routing channels, delay, or a combination of any number of the above. In any case, the cost function should reflect the goals of the placement algorithm.

Finally, after the circuit is placed, the routing procedure completes the layout process by mapping interconnect onto the carrier. The ease with which the routing procedure is able to route the circuit determines the true worth of the placement algorithm. If the placement procedure did a poor job of placement, the routing procedure will not be able to complete the routing, and the user may have to intervene and
finish routing the difficult portion of the circuit. After the routing is finished, the layout process is complete and the circuit is ready for fabrication.

Placement can be achieved using many different methods. The method used in this paper is placement by partitioning, which is based on the availability of a partitioning procedure. The placement by partitioning method divides the carrier into decreasingly smaller blocks of components until each block contains only one component, at which point placement is complete.

Chapter 2 surveys the body of work that has been done in this area over the past years. With the background established, Chapter 3 describes the proposed placement algorithm and the performance-oriented details of its implementation. This is followed by Chapter 4, which reports the results of two basic experiments that were performed in this research. The first was aimed at optimizing the placement algorithm by varying several parameters of the algorithm and comparing the resulting performances. The goal of the second experiment was to gain an insight into the relative performance of the placement algorithm by comparing its performance with that of a simulated annealing
algorithm. The experiments were conducted on actual circuits to ensure the validity of the results. Chapter 5 proposes areas in which this body of work could be further explored, and Chapter 6 concludes the paper.
CHAPTER 2
SURVEY OF PREVIOUS WORK

This chapter is a survey of the body of work that has been created in solving the placement problem. It is important to note that the problem itself is NP-complete and that it is, therefore, infeasible to try to find an optimal solution to the placement problem. Instead, one must create heuristics that attempt to find "good" solutions in a reasonable amount of time. These heuristics can be broken up into constructive and iterative placement algorithms.

Constructive placement algorithms usually start with one or more "seed" components that are already placed and then add other components that are connected to them. The objective is to keep those components that are connected as close as possible to each other so that the total wire length is minimized. Iterative improvement algorithms start with some initial placement and then improve upon the placement in stages. The improvement is most often found by moving a small subset of the components, and is repeated until some "stopping criterion" is met. These two classes of placement algorithms will be discussed further in the following sections.
2.1 Constructive Placement Algorithms

Constructive placement algorithms, also known as constructive initial placement algorithms [Wol78], take an unplaced set of components and their respective nets and assign the components to positions on the carrier. Typical constructive placement algorithms are the cluster growth, force directed, branch-and-bound, and placement by partitioning techniques. The next four sections will briefly introduce each of these classes.

2.1.1 Cluster growth

Cluster growth constructive placement algorithms start with an empty carrier and then pick a component, placing it on the carrier. This process is continued until all the components have been placed. The process above can be broken into two independent functions designated as the select and place functions. The select function chooses the next component to be placed on the carrier, while the place function determines the position to which it should be assigned. The choice of these two functions characterizes a specific cluster growth algorithm.

One such choice is to randomly select a component from those components not already placed and then place it on the carrier in a randomly chosen position. This is referred to as random placement and, as one might suspect, does not yield very good results [Han76]. However,
the algorithm is very fast and is very easy to implement. This method is often used to create an initial placement to act as input to an iterative placement algorithm.

A more "intelligent" algorithm would choose a component based on its connectivity to the already placed components. The cluster development method [Kur65, Han72] is an example of a more intelligent cluster growth algorithm. It chooses the component that is most strongly connected to all of the placed components, and then places the component so that it is as close as possible to those connected components.

A variation of the general cluster growth technique was introduced by Schuler and Ulrich [Sch72] in which they create a binary cluster tree that is used to create a linear placement of the components. The cluster tree is created by continually clustering the two entities which are most connected at any given time, where the entities can be either components or clusters of components. The order in which this clustering is created is then used to establish a linear ordering or placement of the components. The resulting linear placement is then folded onto the carrier to form a two-dimensional placement. Cox and Carroll use this method in their standard transistor array (STAR) automatic cell
placement technique [Cox80] and discuss ways of placing the linear ordering onto the carrier so as to allow for an equal amount of horizontal and vertical routes.

2.1.2 Force directed

The force directed method models the attraction between connected components with force analogies [Qui79]. This can be thought of as modeling the components as masses and the nets as springs and then allowing the system to reach its lowest energy state. If only attractive forces are used, the system will collapse on itself. To avoid this pitfall, repulsive forces are introduced. The force directed algorithm is based on minimizing the energy of the system as calculated from the force functions. With simple attractive forces, the energy function is convex allowing for relatively simple solution. However, the addition of repulsive forces destroys the convexity of the energy function, making the problem much more difficult.

2.1.3 Branch-and-bound algorithms

Branch-and-bound algorithms are able to search the entire solution space by cutting off searches when it is obvious that a better solution is impossible, given the assignments already made [Gil62]. Because this algorithm can perform an exhaustive search of all permutations, it is potentially very time-consuming and is only appropriate for small placement problems.
2.1.4 Placement by partitioning

The fourth constructive technique is placement by partitioning. This algorithm is based on the availability of a partitioning program. The partitioning algorithm is used to divide the carrier into increasingly smaller blocks until each block contains only one component at which point placement is complete [Bre77]. The two most common partitioning algorithms used were proposed by Kernighan and Lin [Ker70] and Fiduccia and Mattheyses [Fid82]. They both take initially random partitions and make iterative improvements upon them using the general concept of min-cut partitioning until they no longer are able to find any improvements. Kernighan and Lin use pairwise exchange on all the components in one pass, and then take the best state during the pass as the initial state for the next pass. Fiduccia and Mattheyses improved upon the Kernighan and Lin approach by eliminating the pairwise exchange and allowing a single component to move across partition boundaries by itself (as opposed to pairwise interchange). Because single component moves were allowed, they had to add some type of balancing mechanism to keep the partition sizes relatively equal. They also arrived at an interesting and efficient way of updating the gain values of candidate moves.
The placement by partitioning method can be considered to be a hierarchical placement method because it begins by partitioning the entire carrier, and then recursively partitions each of the resulting regions until each region contains a single component. Thus, the algorithm works on many different levels of partitioning problems which collectively result in a complete placement of the circuit. The advantage of this hierarchical behavior is that large-grain decisions can be made at the top level, without having to consider small-grain details which are handled at lower levels of the placement process. In this way, placement by partitioning uses a hierarchy of partitioning problems to break a very large placement problem into many smaller ones.

Most of the algorithms used in this area establish a sequence of horizontal and vertical cut-lines that are used as the boundary for two-way partitions. Two of the most popular sequences are Quadrature and Slice/bisection as described by Corrigan [Cor79]. Quadrature alternates between vertical and horizontal cut-lines that divide the current area into two equal halves. Slice/bisection begins with a series of horizontal cuts and then a series of vertical cuts and allows partitions of unequal size.

Dunlop and Kernighan used these basic methods to create a placement procedure for standard-cell VLSI circuits [Dun85]. The interesting advance made in this work was the consideration of components external to the area currently being partitioned. This concept is referred to by
Dunlop as "Terminal Propagation" and creates a bias for the positioning of components that have connections external to the active partition area. This insured that, all other things being equal, the component would be placed closer to the external component to which it is connected. They also derived an algorithm by which a completed placement can be converted to form the rows of the standard-cell carrier.

The advantages of the placement by partitioning method over the other constructive placement algorithms are: 1) it allows nets to be represented as connecting more than two components, 2) it takes into account interconnect information at global level, deferring low-level decisions until later in the placement, 3) it spreads interconnect evenly across the carrier so that routing is relatively easy, and 4) it is highly programmable, allowing the user to select the order and position of the slice lines [Pre88]. In general, these algorithms are computationally expensive but provide good results.

2.2 Iterative Placement Algorithms

Iterative placement algorithms, as mentioned above, take an initial placement and make improvements on it. A typical technique is to use the output of one of the constructive placement algorithms mentioned above as the input to an iterative improvement algorithm. It has been shown that the iterative improvement algorithm will yield better results when its initial placement is better [Mur80]. Therefore, it can be useful
to have both a good constructive and a good iterative placement algorithm for a combined system. There are also systems in which the two classes of algorithms are interlaced on an iteration-by-iteration basis [Sch76].

One advantage of the iterative placement algorithms is that they can be stopped when the solution is satisfactory to the user. These algorithms make proposed changes and then decide whether or not to accept the change. If there is an improvement, the changes are retained; otherwise, the original placement is restored. This process is repeated until some stopping criterion is met. The process can be divided into three steps: 1) choice of components to be moved 2) movement of chosen components to their candidate locations, and 3) evaluation of the new placement. The choice of how each of these steps is performed forms a family of algorithms under this class. A representative set of these algorithms will be discussed in the following sections.

2.2.1 Interchange

In pairwise interchange, each component, in turn, is selected to be the primary component. The primary component is then trial interchanged with every other component until an improvement is found. If an improvement is found, the trial interchange is accepted [Han76]. Neighborhood interchange is similar, except that the primary component
is only trial interchanged with other components in its vicinity [Han76].
In this way, components "migrate" to their final destination over several
iterations.

2.2.2 Force-directed algorithms

The force-directed algorithms are characterized by modeling
interconnect as attractive force vectors (e.g., if i and j are connected,
then i and j have an attraction to each other). Force-directed
interchange uses the sum of these vectors to determine which
components to move and in which direction to move them. In the
simple version, the primary component is trial interchanged with its
neighbors in the direction of the force vector [Han76]. In contrast,
force-directed relaxation allows the primary component to be placed at
its zero-force position (position at which the force vectors are zero) and
the component that it displaces is then used as the primary component.
This process continues until the primary component is placed in an
unoccupied position. After completing one series of moves, the new
placement is evaluated and a decision to keep the moves is made.
Force-directed pairwise relaxation also uses the concept of a zero-force
position. However, in this case, two components must be found such that
their zero-force positions and their current positions are close to each
other (i.e., component A is at position 1 and wants to be at position 2,
while component B is at position 2 and wants to be at position 1). The
pair is exchanged and the placement is evaluated, resulting in the change either being saved or thrown out. In general, the force-directed methods converge more quickly than the interchange algorithms, but do not always find optimal solutions.

2.2.3 Steinberg's algorithm

Steinberg's algorithm [Ste61] linearizes the quadratic assignment model of the placement problem by iterating on sets of components that have no connection between them. Each iteration begins by identifying an independent set of components and removing them from the carrier. The cost of placing each component in each vacant position is then computed. These costs are independent of the possible placement of the other components in the designated independent set; therefore, the problem is one of linear assignment of components to positions. The cycle continues until all independent sets have been removed and reassigned new positions on the carrier.

2.2.4 Simulated annealing

Simulated annealing is a method derived from the physical phenomenon of annealing a material. The annealing process begins by heating the material to its melting point and then slowly cooling the material so that its molecules settle to their lowest energy states. Simulated annealing mimics this phenomenon by creating an artificial temperature value and letting the energy of a placement correspond to
the evaluation function (or cost function) [Kir83]. The algorithm starts with an initial placement and then sets the temperature value high enough so that all moves are accepted regardless of whether they represent improvements or not. Then, after a certain number of successful moves have been made, the temperature is lowered and the process is repeated. By lowering the temperature, the probability of accepting bad moves is decreased. This is done by making the probability of accepting a bad move equal to $e^{-DE/T}$, where $DE$ is the change in the energy of the system, or in this case, the change in the evaluation function, and $T$ is the temperature. Good moves are automatically taken without any calculation being made. The temperature is continuously decreased until a certain number of moves are rejected. At this point, the problem is considered to be "frozen" (no further improvements are likely to be made) and the process terminates.

Simulated annealing is a very time-intensive algorithm, but is much more likely to find a global-minimum than the algorithms mentioned above.
CHAPTER 3
PLACEMENT ALGORITHM

This chapter will introduce the placement algorithm that was developed and discuss some of the performance-oriented details of its implementation. The chapter begins with a discussion of the partitioning algorithm in Section 3.1. The use of global awareness to improve placement quality is the topic of Section 3.2. Section 3.3 introduces the idea of partition problem scheduling and discusses some of the scheduling considerations. The chapter concludes with Section 3.4 which discusses some of the performance-oriented mechanisms used in the implementation of the placement algorithm.

3.1 Partitioning Algorithm

The partitioner is a constructive algorithm which divides the circuit into as many as sixteen planar regions (or partitions) as shown in Figure 2. The regions divide the partition space into rows and columns. The partitioner takes, as input, the number of rows and columns of regions the circuit is to be partitioned into. The user is allowed to specify the number of regions and the aspect ratio of the resulting partition space. The partitioner begins with a data structure that
Figure 2: Spatial Orientation and Numbering of the Regions
represents the components and nets of the circuit, and a list of the components yet to be partitioned, called the free list. Initially, the free list will contain the entire set of components to be partitioned. The partitioner then randomly chooses a component, removes it from the free list and assigns it to the middle-most region of the partition space. This component acts as a "seed" for the growth of the partition. The algorithm is also capable of allowing components to be preassigned to partitions, hence allowing for multiple user-assigned seeds. Having assigned the seed(s), the algorithm begins evaluating which component should be assigned next and to which region it should be assigned. This is accomplished by a procedure which evaluates the "affinity" between every component and each region, and outputs the component which has the maximum affinity for each region. Another procedure arbitrates between these component/region pairs and makes one or more assignments of components to regions. This entire process, with the exception of seed assignment, is then repeated until all the components have been assigned to regions, at which time the partition is complete.

The following sections will describe the way in which seed assignment, affinity calculations, size balancing, and arbitration are performed.
3.1.1 Seed assignment

Various methods of seed assignment are possible in this algorithm. All of them are based on a random selection and assignment of components to regions. The first choice to be made is how many seeds to assign. The number of seeds to be assigned can range from zero to the number of components in the circuit (random partitioning). The three options explored were: 1) assigning every region a single component, 2) assigning one component to the middle-most region, and 3) not assigning any seeds. Intuitively, if the algorithm that is to follow the seed assignment is to perform better than random assignment, it makes sense to leave as much of the circuit unassigned and allow the algorithm that follows it to do its job. This would lead one to believe that not assigning any seeds was the preferred option; however, through some experimentation, it was shown that assigning a seed to the middle-most region made the partitioning more evenly distributed about the center of the partition space. It was also shown that assigning more than one seed resulted in partitions with higher costs. Therefore, the option that was decided on was to randomly choose a component and place it in the middle-most region of the partition space.
3.1.2 Neighborhoods and affinities

This section describes the methods used in calculating the affinity of a component to a particular region. In doing this calculation, there are two basic considerations: 1) what is the component connected to and 2) what is the relative size of the region in question. The first consideration is the topic of this section while the second consideration will be discussed in the section that follows.

The affinity calculation for a given component c and a given region looks at all the components that c is connected to, and determines which neighborhoods these components are in. The neighborhoods give a very simple measurement of the Manhattan distance between the region for which the affinity is being evaluated and the region that the connected components are in. Every region has a set of neighborhoods, which are equivalence classes made up of regions with the same Manhattan distance from the specified region. Figure 3 shows the set of neighborhoods for region number five. Each type of neighborhood is given a weighting factor so that assigning a component to a region in a nearby neighborhood is favored over a region in a far away neighborhood. The algorithm then adds all the weighting factors together to form an affinity factor between the given component and region pair. This process is repeated for every component on the free list and, in turn, every region in the partition space.
Figure 3: Neighborhoods for Region 0
3.1.3 Maintaining balance

This section discusses how the affinity calculation takes into account the need to retain balance between the population of each region. If balance were not considered, the algorithm would simply assign all the components in a circuit to the same region, thereby minimizing the cost of the partition. Therefore, an adjustment to the affinity calculation must be made so that assignments to regions that are already heavily populated will become less desirable. Although several different methods were attempted, the method that proved to be the most successful, in terms of maintaining balance without negative impact on the partitioning process, was termed "linear with ceiling."

The method is termed linear because the affinity of a component to a region is linearly decreasing with the number of components assigned to the region, until the number of components reaches the ceiling. All other things being equal, a component will be placed in the region with the fewest number of components.

The method is said to have a ceiling because a maximum value is placed on the number of components that can be assigned to a region after which, the affinity calculation would produce a large negative number. The large negative number indicates that this assignment would be very undesirable. This large negative number proves to be an adequate deterrent from assignment of components to regions which have already
been assigned the maximum number of components. In this way, balance of the regions is maintained while still allowing connectivity (described in Section 3.1.2) to dominate the majority of assignment decisions.

3.1.4 Arbitration and assignment

This section describes how the partitioning algorithm makes the arbitration and assignment decisions. The arbitration procedure analyzes a list of each of the region's most desired component selections and their respective affinity values. This list is referred to as the "wish-list," and can be thought of as a listing of potential assignments of components to regions. The procedure begins its analysis by finding the highest affinity value of all the regions and making all those potential assignments whose corresponding affinity values are equal to this maximum value. In the case that more than one region has the maximum affinity for the same component, the algorithm arbitrarily chooses one over the other and the loser is not assigned anything. After completing the assignment of the desired components to their regions, the arbitration and assignment procedure is complete. This completes one full iteration of the partitioning algorithm.

The partitioning algorithm then iterates another time, generating another "wish-list" and making more assignments. This process continues until the entire free list is exhausted.
3.2 Global Awareness

Global awareness considers components in the circuit that are not contained in the current partitioning problem space. One of the inherent problems with the placement by partitioning method, in its purest form, is that it completely ignores the position of components that are not in the current partition space. In other words, the placement of the first quarter of a circuit is normally totally ignored by the partitioner when placing the components in the second quarter of the circuit. This ignorance can result in poor partitioning decisions and, in turn, a placement which is very difficult and costly to route. For that reason, global awareness was added to the placement algorithm. However, there are many ways in which global placement can be added to the placement algorithm. The following section discusses the mechanism used to introduce global awareness into the partitioning algorithm. Some possibilities that will be experimentally investigated are also described in the following sections.

3.2.1 Imaginary components

Before discussing the possible global awareness methods, it is important to understand how the global awareness methods were introduced into the partitioning algorithm. The mechanisms used were termed imaginary components and allowed the addition of components to partitions without affecting the apparent number of components in the
partition. In this way, the affinity values could be affected by imaginary components while the balance mechanisms were left unaffected. Hence, by adding an imaginary component to a region, the affinity between the components to which it is connected and the region it is in is increased. This increase is made without negatively affecting the balance mechanism. For example, if a component which is not in the current partition space is known to be close to the upper left-hand corner of the partition space, then an imaginary component can be placed in the upper left-hand corner region to bias the placement of the component nearer to the external component. The use of this technique is the topic of the following two sections.

3.2.2 Considering only placed components

The first type of global awareness to be described considers only those components which have been assigned their final placement coordinates. This is a naive approach because the components that have not been placed (but whose positions are partially constrained) are not considered. Basically, this method considers all the external components that are directly connected to internal components and places an imaginary component in the region closest to the external component. One advantage of this method is that it does not make any assumptions about the position of unplaced components and, therefore, avoids some possibly poor decisions.
3.2.3 Considering all components

The alternative method is to consider all the components regardless of whether they are placed or not. This requires the assumption that all unplaced components will be positioned at the center of their respective regions. Obviously, all the components in a region cannot be placed at the center of the region after placement is complete. Therefore, this assumption may create some poor decisions. However, it does consider all the components instead of only considering the small subset of components that have already been placed.

In the case of unplaced components, the choice of how many imaginary components and which regions to assign them becomes a problem. This problem only arises when the external component is currently positioned next to one of the sides of the partition space, which are designated as zones one, three, five, and seven in Figure 4. Three different methods were developed for assignment of imaginary components to the regions that form the side closest to the external component. The first is to place the imaginary component in the middle-most region. Note that the middle-most region is an arbitrary choice in the case of the four- and sixteen-way partition problems where the sides are two and four regions long, respectively. Thus, another method was created which assigns imaginary components to both of the middle-most regions in these two cases. This method suffers from the
Figure 4: Designation of Zones
problem that it overemphasizes the affinity of the internal components for external components by a factor of two. Both of these methods may also create an unnatural attraction to the center regions on the sides. A third and final method was created which assigns imaginary components to all the regions on a side. This method alleviates any particular attraction for the center of the side in question, but suffers from an overemphasis in the affinity of the internal components for external components, by up to a factor of four. These methods of global awareness will be investigated through experimentation in Chapter 4.

3.3 Scheduling of Partitioning Problems

The placement procedure begins by determining a schedule for the placement process. This schedule is a list (or queue) of the partitioning problems that must be solved in order to complete the placement of the circuit. The determination of how many levels of partitions will be needed and the sizes of the partitions at each level is the topic of Section 3.3.1. The scheduling process must also determine the order in which the partition problems are executed. This decision can affect the way in which the circuit is partitioned and the quality of the placement that results. For that reason, many different options were made available to the user of the placement program that allow the user to determine which scheduling method to use. Three basic considerations were identified and the alternatives were each
implemented in the program and experimentally tested. These three considerations will be the topic of discussion for Sections 3.3.2 through 3.3.4 below.

### 3.3.1 Number and sizes of partitions

The scheduling process begins with the determination of how many levels of partitions will be needed and the sizes of the partitions at each level. This calculation depends on the number of components that are to be positioned on the carrier. The algorithm assumes that a square carrier is the goal of the placement and then determines how long each side should be by taking the square root of the number of components and rounding up to the nearest integer. The number that results is the minimum length of the sides of the square that will accommodate the components. This value is then factored into twos, threes and possibly fours. These numbers correspond to four-way, nine-way and sixteen-way partitions, respectively. When the length of the side is not exactly factorable into two's, three's and four's, all the possible factorizations of values slightly greater than the length are generated and the sequence of factors whose product is the closest to the minimal side length is chosen. This set of factors then represents the partitions that will be made at each level of the hierarchy. For example, the factors two, three, and four would indicate that the carrier will first be partitioned into four regions and then each of those regions will be further partitioned into nine
regions and then into sixteen regions. Upon completion of the lowest level partition, the positions of the components are recorded and the placement is complete. Note that the ordering of the sequence given in the example was arbitrarily chosen to be from smallest to largest. We could just as easily have chosen the sequence four, three, two which would reverse the order of the partitions. The choices for the ordering of partitions in the queue is the topic of the next section.

**3.3.2 Depth-first or breadth-first**

The first scheduling consideration involves the ordering of the hierarchical levels of partitions. The first option is to perform all of the partitions that are on a given level of the placement before performing any lower level partitions. This method is called *breadth-first* and is named after the breadth-first search technique. The other option is to follow one branch of the partitioning hierarchy as many levels down as possible. This means that after the first partition is performed, one region is chosen and further partitioned until it is completely placed even before beginning on any of the other top level partitions. This method is called *depth-first* and is named after the depth-first search technique. Without the presence of global awareness, the two methods perform the same partitions at the same level of the hierarchy (but in a different order) yielding identical placements. However, the
introduction of global awareness destroys the time-independence of the partitioning problems. Therefore, the choice between these two methods, with global awareness active, does result in different partitions.

3.3.3 Ascending or descending partition sizes

The second consideration is the relative ordering of the partition sizes in the partition hierarchy. One option is to make the ordering strictly ascending where the partitions with smaller numbers of regions are followed by those with larger numbers of regions. In other words, the ordering might be: two, three, three, four. This would mean that the top level would correspond to the two and, therefore, be partitioned into four regions. The following levels would then be partitioned into the number of regions designated by the schedule. The alternative is to reverse the ordering by making it strictly descending. The same example would then be ordered: four, three, three, two. This time the top level would be divided into a sixteen-way partition and the lowest level would be a four-way partition.

The two methods will definitely result in varying final placements. The choice can be seen as the difference between making finer-grain decisions first or making larger-grain decisions first. This difference, along with the other considerations discussed in this chapter, will be experimentally investigated in Chapter 4.
3.3.4 Use of sixteen-way partitions

This final section on scheduling considerations describes the use of sixteen-way partitions. The partitioning procedure is capable of partitioning into any number of regions up to a total of sixteen. In order to keep the placement as square as possible, the partitions that are considered are also square. Therefore, the choices are a two-by-two partition (four regions), a three-by-three partition (nine regions) or a four-by-four partition (sixteen regions). The latter of these is the subject of this section. Because a sixteen-way partition can be replaced by a two-level hierarchy of four-way partitions, the need for sixteen-way partitions becomes questionable. When using the sixteen-way partition, the partitioning algorithm is more aware of all the components in the sixteen regions. In the second level of the four-way equivalent problem, the partitioner is only aware of one quarter of the resulting sixteen regions. A possible advantage of the two-level four-way method would be that it delays some of its decisions for a lower level. These two methods are evaluated in Chapter 4.

3.4 Implementation

This section discusses some of the performance-oriented implementation details of the partitioning algorithm. Partitioning algorithms, as a whole, suffer from time-intensive computation. For that reason, special consideration for performance was made when
implementing the partitioning algorithm. One area of interest was the data structures to be used. Another area of performance optimization was in the use of bit-wise operations in the calculation of affinities. Because bit-wise operations are such a low-level operation, they can be performed relatively quickly. A third consideration was free store memory allocation, where a mechanism was created that allocates large blocks of memory from the operating system, and then parcels it off in smaller pieces as needed. The final area of performance optimization was the use of multiple processors to perform affinity calculations. These areas will be discussed in detail in the following sections.

3.4.1 Data structures used

Partitioning of a circuit requires knowledge of the circuit, its components, and their connections. This section will highlight the data structures that were used to represent the circuit and provide that knowledge to the algorithm. There are a total of five data structure types that will be described below.

The component-node is the first data type and appears in Figure 5. It represents a component in the circuit. The component-node identifies the component that it represents, maintains information about which partition it has been assigned to, and also maintains placement information used by the placement algorithm. This data structure has two pointers which link it to the rest of the circuit representation. The
Figure 5: Data Structures
first pointer points to the next component-node and allows for the formation of a linked list of component-nodes containing all the components in the circuit. The second pointer points to a net-list which identifies the nets to which the component is connected. The net-list data structure type will be discussed below.

The net-node data structure type represents a net in the circuit (see Figure 5). Like the component-node data structure type, it, too, is part of a linked list. In this case the linked list is made up of net-nodes and contains all the nets of the circuit. The net-node also has a pointer to a component-list which identifies the components which are connected by the net. The component-list data structure type will be discussed below. In addition, each net-node contains a sixteen-bit unsigned number called the Quads field, which indicates which regions (of the sixteen possible) contain components connected by the net. This information is compactly represented in binary form to reduce memory storage as well as to increase performance. For example, if one of the components connected by this net is assigned to region number five (refer to Figure 2), then there would be a binary 'one' in the fifth bit position of the Quads field. The Quads field is used in the affinity calculation described above to determine the proximity of components to the region in question. Manipulation of the Quads field is the topic of the next section.
The third data structure type is the net-list (see Figure 5). The net-list was already introduced above as identifying the nets to which a component is connected. The net-list data structure forms a third type of linked list which contains pointers to net-nodes. The head of this linked list is pointed to by the respective component-node. Thus, a component identifies the nets that it is connected to. Unlike the component-node and net-node linked lists, there are more than one net-node linked lists. In fact, every component-node has its own net-node list which is made up of pointers to the nets that connect that particular component to the rest of the circuit. Therefore, there may be several net-list data structures in different lists that point to the same net-node.

The fourth data structure type, the component-list, is very similar to the net-list data structure except that it forms a list of pointers to gate-nodes instead of net-nodes (see Figure 5). In this way, the component-list identifies the components which are connected by a net. The component-list data type is also the data structure type used for the free list, which points to all the components that have not been assigned to a region.

The fifth data structure type is the partition-node and is used in solving the placement problem by forming a queue of partitioning problems and sub-problems to be executed. The queue is implemented
with a linked list of partition-nodes, and stores all the values that the
partitioning procedure needs to start a new partitioning problem. Every
time a partition is complete, the next partition problem is removed from
the queue and executed.

3.4.2 Use of bit-oriented operations

This section will discuss the usage of bit-oriented operations to
improve performance in the partitioning procedure. As discussed in the
section above, the Quads field is a sixteen-bit unsigned integer variable,
contained in a net-node, that indicates which of the sixteen regions
contain components connected by the designated net. The Quads field
can be thought of as a series of sixteen Boolean flags, each one
corresponding to a particular region. The flag is set to a Boolean true any
time one or more components connected by the net are assigned to the
region. The exact number of components that are in the region is
irrelevant, because it is assumed that all the components inside a region
can be connected to the components outside the region by a single wire.

The Quads field is maintained and evaluated using bit-oriented
operations. Initially, all the net-nodes' Quads fields are set to zeros.
Then each time a component is assigned to a region, the Quads fields of
all the net-nodes pointed to by the component's net-list are ORed with a
logical "one" in the corresponding bit position. For example, if a
component was being assigned to the eighth region, then a logical "one"
would be ORed with the eighth bit of the Quads fields of those nets which connected the component to the rest of the circuit. In this way, bit-oriented operations are used to maintain a record of the regions that its components have been assigned to.

The bit-oriented operations are also used when calculating the affinity values of components to regions as described above. The affinity values are determined by examining the location of those assigned components that are connected to the component in question. The examination tries to determine how close the component will be to the already assigned components if it is placed in the specified region. This is done using the concept of neighborhoods. Every region has a set of neighborhoods which are equivalence classes of regions with the same Manhattan distance from the specified region. Like the Quads fields, neighborhoods are also represented using sixteen-bit unsigned integer variables with a logical "one" being placed at bit positions that correspond to regions in the neighborhood. Then, to determine how many components on a given net are in a neighborhood, a bit-oriented AND of the neighborhood and the Quads field of the net is performed. The number of logical "ones" in the result of the AND operation is then multiplied by the neighborhood weighting factor. This calculation is
repeated for all six of the region's neighborhoods. Finally, after all the nets that connect the component have been evaluated, a sum for the affinity of the component to the region is obtained.

Because the affinity calculation is repeated so many times, it is very important that the calculation be as quick as possible. The use of bit-oriented operations was one way in which this goal was to be achieved. One of the benefits is that a single operation can evaluate the presence of components in all sixteen regions at once as opposed to evaluating their presence one region at a time. The second benefit is that the bit-oriented operations are a very simple operation for most arithmetic logic units (ALU) to perform and therefore require very few machine cycles. These two effects combine to form a very high performance affinity calculation procedure.

3.4.3 Memory allocation

Memory allocation in C programs is most often performed using the malloc system call, which returns a pointer to a designated block of memory. This section will discuss how this basic memory allocation routine was used to create an internal memory allocation routine which increases performance by reducing the amount of page swapping that is done in memory. The internal memory allocation routine acts as an interface between the standard malloc routine and the internal demands for memory allocation. Page swapping in memory occurs when the active
memory space being used is contained on more than one page of memory. This necessitates the swapping of these pages whenever an access to one of the pages is followed by an access to a different page of memory. The fragmentation of the memory occurs as an effect of other users on the machine requesting memory allocation in between our own requests. The internal allocation routine attempts to reduce page swapping in memory by allocating large blocks of memory at a time and then internally parceling pieces of this block on an as-needed basis. In this way, an attempt is made to maintain spatial locality of the memory that is used in the partitioning and placement procedures. The spatial locality of the allocated memory reduces the chance of time-consuming page swapping from occurring, thus improving performance. Another advantage of the internal memory allocation routine is that it is less time-consuming than calls to malloc.

3.4.4 Parallelization

Recall, from the discussion of bit-oriented operations, that the affinity calculation consumes much of the time required to perform a partitioning problem. This section describes the use of parallel processors to reduce the elapsed time required to complete the necessary affinity calculations, which can be first divided on a per region basis. In other words, a separate calculation can be performed on behalf of each region to determine what the region's most desired component
is. This is done by examining each component on the free list and evaluating the component's affinity to the region. After all the components have been evaluated, the component with the highest affinity for that region is reported. When all regions have determined their most desired component, the arbitration and assignment procedure takes over.

The next step is to assign the per region calculations to processors to be computed in parallel, with their results being returned to the host process. This is accomplished by a process called forking, which allows the creation of multiple identical processes to be spawned by the host process. These processes are then scheduled for execution on processors. Upon completion, control returns to the host process. In this way, each process was given the task of determining the most desired component for its respective region.

Because all the processes work on the same set of shared data, it is important to consider the data conflicts that could occur. For example, if two processes try to write to the same data space at the same time, the resulting value will be unpredictable. Errors also occur when one process writes to a value just before another process reads from it. Fortunately, the partitioning procedure avoids these pitfalls of shared memory multiprocessor programming. This is accomplished in two ways. First, the parallel processes never make changes to the data structure that represents the circuit. Their calculations only require read access to the
circuit data space. This avoids any possible errors in the circuit representation. Second, each process writes to a distinct data space that is not written to by any other process. This write is only performed once at the completion of the procedure when the most desired component is being reported. The write acts as a communique to the host process that then arbitrates between all of the region's desired assignments.

**Load balancing** is the term used to describe the assignment of the work load to processes that will perform them. The load balancing used in the affinity calculation procedure is dynamic, which means that it is determined at run time based upon how many processors are to be used and how quickly each of the calculations is completed. Each of the processes starts its routine by accessing a global shared variable which indicates the next region to be calculated. The access to the global variable also increments the value of the global variable in preparation for access by the next process. The process then does the necessary calculations for the region and reports the result. After reporting the result, the process again accesses the global shared variable and repeats the calculation for that region. The process terminates whenever the region number that the process receives is invalid. After all the processes have terminated, control is returned to the host process which continues the partitioning procedure. This form of load balancing maintains a good balance even in very adverse conditions. By using
several processes to do the work normally done by a single process. the parallel implementation of the affinity calculation procedure shows significant speedups over the single process version.
CHAPTER 4
EXPERIMENTAL RESULTS

This chapter will introduce the experiments that were performed to try to find good parameter settings and to evaluate the relative performance of the placement algorithm. Section 4.1 will identify the source of the test cases that were used in the experimentation. The measures used to evaluate performance will be introduced in Section 4.2. Section 4.3 will discuss the effect of other users on the performance of the Sequent Balance 8000, and Section 4.4 will present the results of the related speedup analysis that was performed. Section 4.5 discusses the parameter variation experiments that were performed to optimize the placement algorithm. Section 4.6 reports on the performance of the optimized placement algorithm across all of the benchmark test cases. Section 4.7 discusses the use of simulated annealing as a benchmark while Section 4.8 reports on the simulated annealing results and how they compare to those generated by the placement algorithm.
4.1 Test Cases Used

The test cases that were used were chosen from among sixteen benchmark test cases distributed for the 1988 International Placement-and-Routing Workshop at the Microelectronics Center at North Carolina (MCNC/SIGDA CAD benchmarks). The benchmarks were originally written in Yet Another Language (YAL) which was developed by Bryan Preas and Ken Roberts. This language offers a great deal of information about the placement of pads and the technology to be used. A translation program was written which took the YAL descriptions and put them into a format that was compatible with the placement algorithm. In the process, pads were treated as components and technology information was omitted. Basically, the translator extracts the net-list from the benchmark test cases.

The subset of the benchmarks used in this paper was chosen based on their size and the impracticality of attaining results on any larger benchmarks. The benchmark test cases that were used are labeled: Primary1, Test01, Test02, Test03, and Test04. These benchmarks contained 833, 10, 1663, 1607, and 1515 components, respectively. These benchmarks were abstracted from actual circuit designs and, therefore, provided realistic test cases for the experiments that were performed.
4.2 Performance Measures

This section will discuss the two performance measures used in the experiments: elapsed time and wiring cost. Elapsed time indicates the total run time of the algorithms. The elapsed time is like a stopwatch which is started after the circuit has been loaded into memory and stopped when the algorithm is complete. The elapsed time is a real-time clock and is reported as part of the output upon completion.

The wiring cost is an estimate of how much interconnect it would require to fully route the circuit based upon the positions of the components in the completed placement. The calculation that is used is often referred to as the half-perimeter bounding-box calculation. This means that the minimal rectangular box is found for each net that encloses all the components that the net connects. The wiring cost of the net is then estimated as one-half the perimeter length of this box. The wiring cost is then the sum of the half-perimeters of the bounding-boxes for the nets in the circuit. This value is calculated after the placement is complete and the elapsed time has been recorded. Together these two performance measures give indicators for the tradeoff between execution time and quality of result.
4.3 Effect of Other Users of Sequent Balance 8000

The multiprocessor used in these experiments is the Sequent Balance 8000 (hence referred to as the Sequent). The Sequent is a multiuser multiprocessor. This means that the eight processors on the machine are utilized in a round-robin order to service the requests of all the users on the machine. The number of users may vary from one to sixty, or more. The users' processes are placed in a queue and eventually assigned to processors for execution. Parallelism is accomplished in the Sequent by allowing a host process to create (fork) several child processes. The child processes are then assigned to processors and executed. However, there is no guarantee that each process created will have a processor completely dedicated to execute it. Depending on the number of processes waiting to be executed, each process may end up with a very small fraction of a processor's time. Under a very heavy load, all the forked processes could end up being executed on a single processor. Obviously, this has a direct effect on the performance of the program and also on the amount of speedup that is achievable. Speedup is the term used for the decrease in execution time which is achieved when the number of processors executing a program is increased. Other users on the machine effectively cut down the number
of free processors, forcing the forked children to share the processors that remain. Therefore, it is not always advantageous to fork the maximum number of processes.

Two measures were taken to ensure that valid data were taken. The first was to perform the experiments between the hours of midnight to six in the morning so that the number of users was minimized. The second measure taken was to perform a speedup analysis, which determined the optimal number of processes to be used. The results of this experiment dictated the number of processes that were to be forked during the remainder of the tests. The results are contained in the next section.

4.4 Speedup Analysis

In order to determine the optimal number of processes to create, a speedup analysis was performed. The results of the speedup analysis are shown in Figure 6. As was explained above in 4.3, the performance of parallel programs on the Sequent is directly affected by other users on the machine. For that reason, optimal performance is not always gained by creating the maximum number of child processes (seven). The presence of more processes than processors can actually hinder performance substantially. The speedup analysis consists of varying the number of processes created and measuring the elapsed time. The
Figure 6: Speedup Analysis
Elapsed Time vs. Number of Processes
analysis was performed at the same time of the day when the experimental runs were to be made, with the hope that the load on the machine would be consistent from day to day.

Looking at the results in Figure 6, it is clear that increasing the number of processes does improve performance. However, there are two interesting points on the plot: one at five processes, and one at seven processes. Upon initial inspection, these discontinuities were believed to be caused by a flux in the user load on the Sequent. However, these tests were repeated again with the same results. We believe that these discontinuities in the otherwise smooth curve are a side-effect of the load balancing mechanism. The test case used in the speedup analysis was Primary1 which is placed using four and sixteen-way partitions. It is important to understand that the performance of the algorithm is determined by the workload of the most heavily loaded process. For instance, when sixteen-way partitions are performed, using four processes means that each process will be assigned four regions. However, using six processes means that each process will be assigned, at most, only three regions. In the case of five processes, however, the maximum number of regions is four, the same number as when four processes are used. Because the maximum number of regions assigned to
any process is the same for four and five processes, only a slight improvement in performance is obtained. This phenomenon also causes the deflection in the curve that occurs going from six to seven process.

As a result of the speedup analysis, the number of processes chosen for use in the remainder of the experiments was six.

4.5 Variation of Placement Parameters

This section presents the results of the optimization experiments that were performed to determine the optimal parameter settings for the placement algorithm. The parameters that were varied were global awareness (five options), breadth-first or depth-first scheduling, ascending or descending scheduling, and the sixteen-way partition option. These parameters are explained in Sections 3.2 and 3.3. In these experiments, an exhaustive testing of all the possible combinations of the parameters was made. There were a total of forty distinct combinations of the parameters. The test case used was Primary1.

In order to report the results of these experiments in a compact fashion, a code system was created to identify the parameter combinations. These codes are four letters long. The first letter of the code indicates the global awareness method that was used (N - No Global Awareness Method, P - "Placed Only", M - "All - Middlemost", T - "All - Middle One or Two", W - "All - Whole Row/Column"). The second letter indicates whether the breadth-first or the depth-first scheduling method
was used (D - Depth-first, B - Breadth-first). The third letter indicates whether ascending or descending order was used in the scheduling process (A - Ascending, D - Descending). The last letter indicates if sixteen-way partitions were allowed (Y - Yes, they were allowed, N - No, they were not allowed). The presence of an underscore indicates that the data points above the label vary along this parameter. The underscore is used as a place-holder.

The results of the experiments are shown in Tables 1 through 5 and are grouped by global awareness techniques. These results are also graphically displayed in Figures 7 through 14. Each pair of figures shows the forty data points grouped by different parameters. The data points that share a horizontal axis position also share parameter settings with the exception of the parameter that is being compared. In this way, one can analyze the effects of each of the parameters individually. Note that the leading underscore is not used when the global awareness parameter is varied in Figures 7 and 8.

Looking at Figures 7 and 8, one sees that the addition of the global awareness techniques, as a whole, does make a substantial improvement on wiring cost. One can also see that when only placed components are considered, the depth-first option provides a much better placement than the breadth-first option. This is due to the fact that the depth-first
Table 1: Placement Results for Testcase Primary1 Without Global Awareness

<table>
<thead>
<tr>
<th>ID#</th>
<th>Depth/Breadth</th>
<th>Ascend/Descend</th>
<th>Allow 16-way</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Depth</td>
<td>Ascend</td>
<td>Yes</td>
<td>12486</td>
<td>1214.580</td>
</tr>
<tr>
<td>2</td>
<td>Depth</td>
<td>Ascend</td>
<td>No</td>
<td>13205</td>
<td>894.300</td>
</tr>
<tr>
<td>3</td>
<td>Depth</td>
<td>Descend</td>
<td>Yes</td>
<td>10639</td>
<td>1981.850</td>
</tr>
<tr>
<td>4</td>
<td>Depth</td>
<td>Descend</td>
<td>No</td>
<td>13205</td>
<td>912.090</td>
</tr>
<tr>
<td>5</td>
<td>Breadth</td>
<td>Ascend</td>
<td>Yes</td>
<td>12486</td>
<td>1219.350</td>
</tr>
<tr>
<td>6</td>
<td>Breadth</td>
<td>Ascend</td>
<td>No</td>
<td>13205</td>
<td>895.080</td>
</tr>
<tr>
<td>7</td>
<td>Breadth</td>
<td>Descend</td>
<td>Yes</td>
<td>10639</td>
<td>1972.690</td>
</tr>
<tr>
<td>8</td>
<td>Breadth</td>
<td>Descend</td>
<td>No</td>
<td>13205</td>
<td>893.860</td>
</tr>
</tbody>
</table>

Table 2: Placement Results for Testcase Primary1 Using "Placed Only" Global Awareness Technique

<table>
<thead>
<tr>
<th>ID#</th>
<th>Depth/Breadth</th>
<th>Ascend/Descend</th>
<th>Allow 16-way</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Depth</td>
<td>Ascend</td>
<td>Yes</td>
<td>10270</td>
<td>1122.830</td>
</tr>
<tr>
<td>10</td>
<td>Depth</td>
<td>Ascend</td>
<td>No</td>
<td>10819</td>
<td>872.240</td>
</tr>
<tr>
<td>11</td>
<td>Depth</td>
<td>Descend</td>
<td>Yes</td>
<td>9381</td>
<td>1975.860</td>
</tr>
<tr>
<td>12</td>
<td>Depth</td>
<td>Descend</td>
<td>No</td>
<td>10819</td>
<td>918.190</td>
</tr>
<tr>
<td>13</td>
<td>Breadth</td>
<td>Ascend</td>
<td>Yes</td>
<td>12039</td>
<td>1207.480</td>
</tr>
<tr>
<td>14</td>
<td>Breadth</td>
<td>Ascend</td>
<td>No</td>
<td>13053</td>
<td>896.460</td>
</tr>
<tr>
<td>15</td>
<td>Breadth</td>
<td>Descend</td>
<td>Yes</td>
<td>10466</td>
<td>1973.910</td>
</tr>
<tr>
<td>16</td>
<td>Breadth</td>
<td>Descend</td>
<td>No</td>
<td>13053</td>
<td>894.410</td>
</tr>
</tbody>
</table>
### Table 3: Placement Results for Testcase Primary1 Using "All - Middlemost" Global Awareness Technique

<table>
<thead>
<tr>
<th>ID#</th>
<th>Depth/Breadth</th>
<th>Ascend/Descend</th>
<th>Allow 16-way</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Depth</td>
<td>Ascend</td>
<td>Yes</td>
<td>10002</td>
<td>1109.250</td>
</tr>
<tr>
<td>18</td>
<td>Depth</td>
<td>Ascend</td>
<td>No</td>
<td>9706</td>
<td>857.250</td>
</tr>
<tr>
<td>19</td>
<td>Depth</td>
<td>Descend</td>
<td>Yes</td>
<td>9166</td>
<td>1951.720</td>
</tr>
<tr>
<td>20</td>
<td>Depth</td>
<td>Descend</td>
<td>No</td>
<td>9706</td>
<td>857.810</td>
</tr>
<tr>
<td>21</td>
<td>Breadth</td>
<td>Ascend</td>
<td>Yes</td>
<td>10186</td>
<td>1127.970</td>
</tr>
<tr>
<td>22</td>
<td>Breadth</td>
<td>Ascend</td>
<td>No</td>
<td>10215</td>
<td>870.200</td>
</tr>
<tr>
<td>23</td>
<td>Breadth</td>
<td>Descend</td>
<td>Yes</td>
<td>9172</td>
<td>1952.690</td>
</tr>
<tr>
<td>24</td>
<td>Breadth</td>
<td>Descend</td>
<td>No</td>
<td>10215</td>
<td>869.820</td>
</tr>
</tbody>
</table>

### Table 4: Placement Results for Testcase Primary1 Using "All - Middle One or Two" Global Awareness Technique

<table>
<thead>
<tr>
<th>ID#</th>
<th>Depth/Breadth</th>
<th>Ascend/Descend</th>
<th>Allow 16-way</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Depth</td>
<td>Ascend</td>
<td>Yes</td>
<td>9697</td>
<td>1108.620</td>
</tr>
<tr>
<td>26</td>
<td>Depth</td>
<td>Ascend</td>
<td>No</td>
<td>9461</td>
<td>866.620</td>
</tr>
<tr>
<td>27</td>
<td>Depth</td>
<td>Descend</td>
<td>Yes</td>
<td>9270</td>
<td>1952.320</td>
</tr>
<tr>
<td>28</td>
<td>Depth</td>
<td>Descend</td>
<td>No</td>
<td>9461</td>
<td>866.970</td>
</tr>
<tr>
<td>29</td>
<td>Breadth</td>
<td>Ascend</td>
<td>Yes</td>
<td>9954</td>
<td>1098.950</td>
</tr>
<tr>
<td>30</td>
<td>Breadth</td>
<td>Ascend</td>
<td>No</td>
<td>10120</td>
<td>868.300</td>
</tr>
<tr>
<td>31</td>
<td>Breadth</td>
<td>Descend</td>
<td>Yes</td>
<td>9138</td>
<td>1950.290</td>
</tr>
<tr>
<td>32</td>
<td>Breadth</td>
<td>Descend</td>
<td>No</td>
<td>10120</td>
<td>869.190</td>
</tr>
</tbody>
</table>
Table 5: Placement Results for Testcase Primary1 Using "All - Whole Row/Column" Global Awareness Technique

<table>
<thead>
<tr>
<th>ID#</th>
<th>Depth/Breadth</th>
<th>Ascend/Descend</th>
<th>Allow 16-way</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>Depth</td>
<td>Ascend</td>
<td>Yes</td>
<td>9776</td>
<td>1132.760</td>
</tr>
<tr>
<td>34</td>
<td>Depth</td>
<td>Ascend</td>
<td>No</td>
<td>9472</td>
<td>868.040</td>
</tr>
<tr>
<td>35</td>
<td>Depth</td>
<td>Descend</td>
<td>Yes</td>
<td>9547</td>
<td>1965.430</td>
</tr>
<tr>
<td>36</td>
<td>Depth</td>
<td>Descend</td>
<td>No</td>
<td>9472</td>
<td>877.400</td>
</tr>
<tr>
<td>37</td>
<td>Breadth</td>
<td>Ascend</td>
<td>Yes</td>
<td>10075</td>
<td>1114.210</td>
</tr>
<tr>
<td>38</td>
<td>Breadth</td>
<td>Ascend</td>
<td>No</td>
<td>10124</td>
<td>868.360</td>
</tr>
<tr>
<td>39</td>
<td>Breadth</td>
<td>Descend</td>
<td>Yes</td>
<td>9266</td>
<td>1953.080</td>
</tr>
<tr>
<td>40</td>
<td>Breadth</td>
<td>Descend</td>
<td>No</td>
<td>10124</td>
<td>873.520</td>
</tr>
</tbody>
</table>
Figure 7: Comparison of Wiring Costs for Various Global Awareness Techniques

Figure 8: Comparison of Elapsed Times for Various Global Awareness Techniques
Figure 9: Comparison of Wiring Costs for Breadth-first and Depth-first Scheduling Techniques

Figure 10: Comparison of Elapsed Times for Breadth-first and Depth-first Scheduling Techniques
Figure 11: Comparison of Wiring Costs for Ascending and Descending Scheduling Techniques

Figure 12: Comparison of Elapsed Times for Ascending and Descending Scheduling Techniques
Figure 13: Comparison of Wiring Costs with and Without Sixteen-way Partitions

Figure 14: Comparison of Elapsed Times with and Without Sixteen-way Partitions
option gives the "Placed Only" global awareness method more information at an early point in the placement process than the breadth-first option because the depth-first option places more components earlier.

The experiments were successful in narrowing down the field of possible parameter settings from forty to four. The four near-optimal results were those generated by the following parameter settings: MDDY, MBDY, TDDY, and TBDY. Note the common parameter settings in these cases were the descending-order scheduling technique and the use of sixteen-way partitions. The global awareness parameter was also narrowed down from five possibilities to two: "All - Middlemost" and "All - Middle One or Two." Further discrimination between the remaining parameter combinations requires more extensive testing using a wide range of test case.

4.6 Performance of Optimized Placement Algorithm

Having narrowed down the field of possible optimal parameter combinations using only one test case, it was necessary to expand the number of test cases. Therefore, the candidates were tested again using benchmark test cases Test01, Test02, Test03, and Test04. The results are tabulated below in Tables 6 through 9. The results for Test02, Test03, and Test04 are displayed in Figure 15.
Table 6: Results of MDDY Version of Placement Algorithm
on Benchmark Testcases

<table>
<thead>
<tr>
<th>Benchmark Testcase Name</th>
<th># Gates</th>
<th># Nets</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary1</td>
<td>833</td>
<td>904</td>
<td>9166</td>
<td>1951.720</td>
</tr>
<tr>
<td>Test01</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>0.500</td>
</tr>
<tr>
<td>Test02</td>
<td>1663</td>
<td>1795</td>
<td>25998</td>
<td>7848.260</td>
</tr>
<tr>
<td>Test03</td>
<td>1607</td>
<td>1632</td>
<td>25495</td>
<td>8529.890</td>
</tr>
<tr>
<td>Test04</td>
<td>1515</td>
<td>1700</td>
<td>23248</td>
<td>7741.520</td>
</tr>
</tbody>
</table>

Table 7: Results of MBDY Version of Placement Algorithm
on Benchmark Testcases

<table>
<thead>
<tr>
<th>Benchmark Testcase Name</th>
<th># Gates</th>
<th># Nets</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary1</td>
<td>833</td>
<td>904</td>
<td>9172</td>
<td>1952.690</td>
</tr>
<tr>
<td>Test01</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>0.510</td>
</tr>
<tr>
<td>Test02</td>
<td>1663</td>
<td>1795</td>
<td>26098</td>
<td>7889.270</td>
</tr>
<tr>
<td>Test03</td>
<td>1607</td>
<td>1632</td>
<td>25672</td>
<td>8547.800</td>
</tr>
<tr>
<td>Test04</td>
<td>1515</td>
<td>1700</td>
<td>23103</td>
<td>7634.970</td>
</tr>
</tbody>
</table>
Table 8: Results of TDDY Version of Placement Algorithm on Benchmark Testcases

<table>
<thead>
<tr>
<th>Benchmark Testcase Name</th>
<th># Gates</th>
<th># Nets</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary1</td>
<td>833</td>
<td>904</td>
<td>9270</td>
<td>1952.320</td>
</tr>
<tr>
<td>Test01</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>0.510</td>
</tr>
<tr>
<td>Test02</td>
<td>1663</td>
<td>1795</td>
<td>25770</td>
<td>7969.420</td>
</tr>
<tr>
<td>Test03</td>
<td>1607</td>
<td>1632</td>
<td>25256</td>
<td>8762.150</td>
</tr>
<tr>
<td>Test04</td>
<td>1515</td>
<td>1700</td>
<td>23243</td>
<td>7562.490</td>
</tr>
</tbody>
</table>

Table 9: Results of TBDY Version of Placement Algorithm on Benchmark Testcases

<table>
<thead>
<tr>
<th>Benchmark Testcase Name</th>
<th># Gates</th>
<th># Nets</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary1</td>
<td>833</td>
<td>904</td>
<td>9138</td>
<td>1950.290</td>
</tr>
<tr>
<td>Test01</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>0.510</td>
</tr>
<tr>
<td>Test02</td>
<td>1663</td>
<td>1795</td>
<td>25817</td>
<td>7990.330</td>
</tr>
<tr>
<td>Test03</td>
<td>1607</td>
<td>1632</td>
<td>25338</td>
<td>8540.670</td>
</tr>
<tr>
<td>Test04</td>
<td>1515</td>
<td>1700</td>
<td>23210</td>
<td>7499.110</td>
</tr>
</tbody>
</table>
Figure 15: Comparison of Near-Optimal Placement Parameter Settings
The results do not conclusively identify a single optimal parameter setting. For Primary1, the best wiring cost is produced by TBDY, while TDDY produces the best result for Test02 and Test03. Adding to the confusion, MBDY produces the best result for Test04. These results indicate that there may not be any overall optimal parameter setting for every test case. However, the TBDY version consistently produces the best or second best result. This parameter setting was, therefore, used in the experiments that compared the placement algorithm to the simulated annealing algorithm discussed in the next section.

4.7 Simulated Annealing As a Benchmark Algorithm

To attain an understanding of the relative performance of the placement algorithm, the same test cases were run through the simulated annealing process which was described in Chapter 3. The implementation used the same data structures and timing mechanisms as the placement algorithm and was run on the Sequent Balance 8000. In terms of elapsed time, the simulated annealing algorithm is known to be extremely poor. However, the simulated annealing results provide a lower bound for the wiring cost for each of the test cases. The wiring costs produced by the placement algorithm can then be compared to those produced by the simulated annealing algorithm, providing a relative measure of the quality of the result.
4.8 Simulated Annealing Results vs. Our Results

This section presents the results of the simulated annealing experiments that were run on the five benchmark test cases and compares the results to those obtained using the optimized placement algorithm. The wiring cost and elapsed time data for the simulated annealing are presented in Table 10. The annealing process can be graphically displayed in a wiring cost vs. elapsed time plot. Plots of this nature were created for each of the test cases and are presented in Figures 16 through 20. The results of the simulated annealing process and of the MBDY version of the placement algorithm are shown together in Figures 21 and 22.

These plots show that the placements produced by the simulated annealing process have lower wiring costs than that of the placement algorithm. However, it is important to note the substantial difference in the elapsed time. The simulated annealing process has long been known for its ability to arrive at a global minimum. However, the time it takes for these results is sometimes impractical. In this case, the simulated annealing process ran more than 63 hours for two of the test cases. This amount of time is often impractical, and can be very expensive. What makes the placement algorithm presented in this paper worthwhile is that it finds a "good" placement solution in a reasonable amount of time.
Table 10: Results of Simulated Annealing on Benchmark Testcases

<table>
<thead>
<tr>
<th>Benchmark Testcase Name</th>
<th># Gates</th>
<th># Nets</th>
<th>Wiring Cost</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary1</td>
<td>833</td>
<td>904</td>
<td>7005</td>
<td>10198.660</td>
</tr>
<tr>
<td>Test01</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>40.250</td>
</tr>
<tr>
<td>Test02</td>
<td>1663</td>
<td>1795</td>
<td>18873</td>
<td>228532.720</td>
</tr>
<tr>
<td>Test03</td>
<td>1607</td>
<td>1632</td>
<td>16573</td>
<td>159859.630</td>
</tr>
<tr>
<td>Test04</td>
<td>1515</td>
<td>1700</td>
<td>20910</td>
<td>229346.040</td>
</tr>
</tbody>
</table>

Figure 16: Results of Simulated Annealing on Testcase Primary1
Wiring Cost vs. Elapsed Time (seconds)
Figure 17: Results of Simulated Annealing on Testcase Test01
Wiring Cost vs. Elapsed Time (seconds)

Figure 18: Results of Simulated Annealing on Testcase Test02
Wiring Cost vs. Elapsed Time (seconds)
Figure 19: Results of Simulated Annealing on Testcase Test03
Wiring Cost vs. Elapsed Time (seconds)

Figure 20: Results of Simulated Annealing on Testcase Test04
Wiring Cost vs. Elapsed Time (seconds)
Figure 21: Comparison of Wiring Costs for TBDY and Simulated Annealing Placement Algorithms

Figure 22: Comparison of Elapsed Times for TBDY and Simulated Annealing Placement Algorithms
CHAPTER 5
FUTURE RESEARCH

This chapter gives some directions in which this research could be expanded. The first section deals with the possibility of varying yet another parameter, i.e., the neighborhood weighting factors. The next two sections discuss the introduction of two more user inputs that would make the algorithm a more practical placement tool. The fourth section proposes a sixth global awareness method that would require the use of fractional imaginary components. The fifth section discusses the use of this placement algorithm as the first in a two-phase placement algorithm. The last section identifies a possible performance improvement that could result from more efficient forking of parallel processes.

5.1 Variation of Neighborhood Weighting Factors

The neighborhood weighting factors determine the affinity of components to regions near connected components. During the course of the experiments, these weighting factors were set in a linearly increasing fashion, such that components would be attracted to regions which contain connected components. These weighting factors do not appear to be as significant as the parameters tested above. However, this
does present another possibility for parametrization. One possibility is to change the slope of these weighting factors so that, for example, being in the same region as the connected component is a factor of ten times more desirable than being in the region right next to it.

One of the problems with the current affinity calculation is that it counts some wire lengths twice. This happens because the affinity calculation considers the distance between the candidate component and each of the placed components separately. Figure 23 demonstrates a situation in which this problem surfaces. In this case, the calculation would count both of the lengths shown instead of taking the longer of the two and assuming that the component in the middle would be able to share the interconnecting signal line. Thus, the calculation would interpret the additional wire length to be five units instead of three units. Changing the affinity calculation to take this effect into account would certainly benefit the placement algorithm.

5.2 Aspect Ratio As User Input

A second area for further work on this algorithm would be to allow the user to input the desired aspect ratio of the resulting placement. The current implementation of the algorithm assumes a one-to-one aspect ratio and produces a square placement. Allowing the user to specify the aspect ratio would make the algorithm more flexible, allowing the user to fit the placement to his technology constraints. The change

73
Figure 23: Problem with "Affinity" Calculation
would have to be made in the scheduling algorithm which assigns the row and column lengths to the top-level partition problem. This change should not affect the performance of the algorithm and would make it much more flexible.

5.3 Preplaced Components

Another change in the algorithm that would enhance its use as a placement tool would be to allow the placement of certain components by the user prior to the algorithm's execution. This would allow the user to place some components at the edge of the carrier to act as contact pads. The change would also allow the user to place components in the middle of the carrier for heat dissipation or other technology-dependent reasons.

5.4 Fractional Imaginary Components

This section deals with a sixth global awareness strategy which would require the use of fractional imaginary components. The concept of imaginary components was introduced in Section 3.3.1. The new strategy would be most similar to the "All - Whole Row/Column" technique, which places imaginary components on the entire length of the side which is closest to the external component. The difference would be that only a fraction of a component, equal to the inverse of the number of regions on that side, would be placed in each region on the side. In this way, the total pull of the imaginary components along the axis perpendicular to that side would be equal to that of one component.
Meanwhile, there would be no pull in either direction along the axis parallel to that side. This solution would combine the advantages of all three of the global awareness strategies that consider all components. Unfortunately, it is not currently possible to implement this solution given the current bit-oriented implementation of the affinity calculation. Therefore, the mechanisms of the partitioning algorithm would have to be completely revised to allow for fractional imaginary components. We suspect that this global awareness strategy would produce better results on a more consistent basis than the other strategies.

5.5 Use As Input to Iterative Placement Algorithm

Although the placement algorithm produces a good result, a better placement may be possible by using the results of this placement algorithm as input to an iterative placement algorithm. It has been shown that an iterative improvement algorithm will yield better results when its initial placement is better [Mur80]. It is possible that this placement algorithm combined with a good iterative placement algorithm could bring the placement even closer to the values produced by the simulated annealing process.
CHAPTER 6
CONCLUSIONS

This paper has introduced a new placement algorithm, which uses a constructive sixteen-way planar partitioner which has been optimized for performance through several methods including parallel processing. The algorithm was optimized through parametric testing, which led to a better understanding of how scheduling and global awareness can affect placement by partitioning algorithms. The placement algorithm's performance was then compared with that of the simulated annealing algorithm. The comparison showed that while the placement algorithm does not provide the "best" results it does provide "good" results in a short time.

Placement algorithms as a whole can be distributed along a quality vs. time axis where the quality of the result is directly related to the time invested. These two axes represent the two major goals in the computer-aided design arena. The first is to provide high quality results, and the second is to provide those results in a short time. The placement algorithm proposed in this paper attempts to provide a happy medium
between these two conflicting demands. It is up to the user of these placement tools to decide what the relative weights are on each of these demands, and then to choose the algorithm that best suits his needs.
REFERENCES


Appendix: Placement Algorithm in C code
Written for the Sequent Balance 8000
A Constructive Placement by Partitioning
Algorithm Using A Performance-Oriented Sixteen-way Planar Partitioning Algorithm
Written for the degree of Master of Science in Electrical Engineering
James Patrick Stapleton
University of Illinois at Urbana-Champaign
Started: 5/26/89
Completed: 7/25/89
NOTE: Must be compiled with: 'cc filename.c -lpps -lm'
NOTE: This is machine dependent code and must be run on Sequent

#include <stdio.h>
#include <math.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
#include "sys/h/types.h"
#include <time.h>
#include "sys/h/timeb.h"

/* DON'T CHANGE THESE ! ! ! ! */

#define unshort unsigned short
#define MAXINT 2.147483647E09
#define TRUE 1
#define FALSE 0
#define NOOWNER -1
#define NOPOS -1
#define JUSTLIN 0 /* constants for BALANCE METHOD software switch */
#define JUSTEXP 1
#define CEILLIN 2
#define CEILEXP 3
#define ALL 0 /* constants for SEED METHOD software switch */
#define MIDDLE 1
#define NONE 2
#define NOGLOBMETH 0 /* constants for GLOBAL METHOD software switch */
#define ONLYPLACED 1
#define ALL_MIDDLE 2
#define ALL_MIDNO 3
#define ALL_WHOLE 4
#define BREADTH 0 /* constants for QORDER software switch */
#define DEPTH 1
#define ASCEND 0 /* constants for DIVORDER software switch */
#define DESCEND 1
#define CENTER 0 /* constants used in set_quads */
#define SIDE 1
#define BOTTOM 2
#define CORNER 3

/***********************/
```c
#define MEMBLOCK 10240
#define FANOUT 5
#define MAXDIV 20
#define MAX_X 100
#define MAX_Y 100

/* Structure definitions */
/*•**********************************************************************/
struct G_node {
    struct G_node *Next; /* Structure that represents a gate */
    int Indx; /* Next points to the next gate in the */
    int Owner; /* linked list. Nets points to a sub- */
    short XPos; /* list of pointers to connected nets */
    short YPos; /* Owner designates the quadrant the */
    short Placed; /* gate was assigned to. */
    struct N_list *Nets; /* Placed is a boolean */
};
typedef struct G_node *G_pter; /* Define a pointer type for G_node */

struct N_list {
    struct N_list *Next; /* List of pointers to net nodes */
    struct N_node *Nets;
};
typedef struct N_list *Nls_pt; /* Define a pointer type for N_list */

struct N_node {
    struct N_node *Next; /* Structure that represents a net */
    unshort Quads; /* Gate points to a list of pointers */
    struct G_list *Gate; /* to gates that are on this net. */
};
typedef struct N_node *N_pter; /* Quads is a 16-bit word that indicates */
/* which quadrants have been assigned */
/* gates that are connected by this net. */

struct G_list {
    struct G_list *Next; /* List of pointers to gate nodes */
    struct G_node *Gate;
};
typedef struct G_list *Gls_pt; /* Define a pointer type for G_list */

struct Q_node {
    short type; /* Q_node represents a partition problem*/
    int xhi; /* that is in the work queue. Q_node */
    int xlo; /* holds all the information needed to */
    int yhi; /* perform the partition. */
    int ylo;
    short nxrows;
    short nocols;
    int numgates;
    float sf[16];
    short div[MAXDIV];
    Gls_pt Gatehead;
    struct Q_node *Next;
};
```
typedef struct Q_node *Q_ptr; /* Define a pointer type for Q_node */

/* Timing structures, and variables */

struct lifetime {
    double startTimeb;
};

struct lifetime BeginTime;

double DeltaT;

/**************************************************************************/

/* Weighting Factors - can be changed to yield different results */

shared int WF[5] = {10, 8, 6, 4, 2}; /* Weighting Factors for neighbors */
shared float BMFCeil = 1.000; /* Balance Factor for Ceiling method */
shared float SF[16] = {
    1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0};

/**************************************************************************/

/* Simulated annealing parameters */

int FAILMULT = 158; /* FAILMULT*total_g attempts w/o success is fail*/
int ACCEPTMULT = 10; /* ACCEPTMULT*total_g accepts is success */
int CONSECFAIL = 3; /* Number of consecutive failures before quitting*/
float COOLDOWN = 0.90; /* determines temperature decrease for annealer */
float HEATUP = 2.00; /* determines temperature increase for melting */

/**************************************************************************/

/* Size parameter to change the number of quadrants to partition into */

shared int NROWS = 4; /* number between 1 and 4 */
shared int NCOLS = 4; /* number between 1 and 4 */

/**************************************************************************/

/* Variables that determine the methods used in the algorithm */

int BALMETHOD = CEILLIN; /* choose from JUSTLIN, JUSTEXP, CEILLIN, CEILLEXP */
int SEEDMETHOD = MIDDLE; /* choose from MIDDLE, ALL, NONE */
int GLOBALMETHOD = ONLyPLACED; /* choose from NOGLOBMETH, ONLyPLACED, */
                        /* ALL_MIDDLE, ALL_MIDTWO, ALL_WHOLE */
int QORDER = BREADTH; /* choose from BREADTH, DEPTH */
int DIVORDER = DESCEND; /* choose from ASCEND, DESCEND */
int DIVWITH4 = FALSE; /* allow 16-way partitions or just 9-way4-way */
int ANNEAL = FALSE; /* perform simulated annealing on result */
int NUMPROCS = 6; /* determines number of processors to use */

/**************************************************************************/

/* Lookup Table */
shared unshort BIT(16) = {32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1};

shared unshort MASK(16)[5] = {
  {0x8000, 0x4800, 0x2480, 0x136C, 0x0013},
  {0x4000, 0x2400, 0x1A40, 0x01BE, 0x0001},
  {0x2000, 0x1200, 0x0820, 0x08D7, 0x0008},
  {0x1000, 0x0410, 0x0212, 0x0863, 0x000C},
  {0x0800, 0x0840, 0x0428, 0x0316, 0x0001},
  {0x0400, 0x04A0, 0x01A4, 0x010B, 0x0000},
  {0x0200, 0x0252, 0x0585, 0x080D, 0x0000},
  {0x0100, 0x0121, 0x0242, 0x0C86, 0x0008},
  {0x0080, 0x0048, 0x0824, 0x6313, 0x1000},
  {0x0040, 0x00A4, 0x04AA, 0xB101, 0x0000},
  {0x0020, 0x0025, 0x2585, 0x8008, 0x0000},
  {0x0010, 0x0012, 0x1242, 0x6C8C, 0x8000},
  {0x0008, 0x0004, 0x0842, 0xC631, 0x1300},
  {0x0004, 0x004A, 0x04A1, 0xEB10, 0x1000},
  {0x0002, 0x0025, 0x0218, 0x7D80, 0x8000},
  {0x0001, 0x0012, 0x0124, 0x36C8, 0xC800}
};

shared unshort NBIT(256) = {
  0, 1, 1, 2, 1, 2, 3, 1, 2, 3, 2, 3, 3, 4,
  1, 2, 2, 3, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
  2, 3, 3, 4, 3, 3, 4, 2, 3, 3, 4, 3, 4, 5, 5,
  1, 2, 3, 3, 3, 3, 4, 2, 3, 3, 4, 3, 4, 5, 5,
  2, 3, 3, 4, 3, 4, 4, 3, 3, 4, 4, 5, 5, 5, 6,
  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 5, 6,
  3, 3, 4, 5, 4, 5, 6, 4, 5, 6, 5, 5, 5, 5, 6,
  1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 5, 6,
  2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 5, 6,
  3, 3, 4, 4, 4, 5, 6, 4, 5, 6, 6, 7, 5, 5, 6,
  3, 3, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 6, 6, 6,
  3, 4, 4, 5, 4, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7,
  3, 4, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7,
  3, 4, 5, 5, 6, 6, 6, 7, 5, 5, 6, 6, 7, 7, 7, 8
};

/********************************************************************************
 * Variable declarations
 *********************************************************************************/
shared G_ptr gate_head;
shared N_ptr nets_head;
shared Gls_pt free_head;
shared Q_ptr Q_head;
shared Q_ptr Q_tail;
shared char* freestart;
shared char* freend;
shared int numalloc = 0;
shared long total_alloc=0;

shared int nbers;
shared int total_g;
shared int total_n;
shared int numgates;

shared int QP[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; /* #gates in each quad */
shared int QPmax = 0; /* Max of QP values */
shared int QPceil;
shared struct wish_pair {
    int INDX;
    double AFFIN;
} wish_list[16]; /* Quad wish list */

shared G_ptr gate_at[MAX_X][MAX_Y]; /* array of pointers corresponding to x,y coordinates of gates */

shared int length_x, length_y; /* size of the placement grid */

int setprocs_ret;

void m_fork(), m_kill_procs(), inl_gate_list(),
inl_net_list(), link_nodes(), readin(),
rem_free(), assign_seeds(),
max_affinity(), output_results(),
arbitrate(), set_ceiling(),init_Q(),
timeStart(), unassign_all(), assign_params(), set_quads(),
print_placement(),sim_annealing(),rand_place();

/**************************************************
*/
/* get_input reads in the input string and returns an integer */
/* **************************************************/

int get_input()
{
    int c, n;
    n = 0;
    while ((c = getchar()) == ' ' || c == '\n' || c == '\t');
    while (c >= '0' && c <= '9') {
        n = (c-48)+(n*10);
        c = getchar();
    }
    return(n);
}

/******************
*/
/* timeStart starts the millisecond timer by calling ftime */
/* ******************/

void timeStart(t)
    struct lifeTime *t;
{
    struct timeb tmb;
    ftime(&tmb);
    t->startTimeb = (double)tmb.time*1000.0 + (double)tmb.millitm;
}

/******************
*/
/* timeDelta returns the number of milliseconds since the timeStart call */
/* ******************/

double timeDelta(t)
    struct lifeTime *t;
{
    struct timeb tmb;
    ftime(&tmb);
    return( (double) ((double)tmb.time*1000.0 + (double)tmb.millitm) -
            (t->startTimeb) ) / 1000.0;
}
/************************************************************************
/* numbits returns the number of one bits in binary rep. of n */
/************************************************************************
int
numbits(n)
unshort n;
{
    int sum = 0;
    unshort temp;
    temp = n;
    temp = temp>>8;
    sum += NBITS[temp];
    temp = n;
    temp = temp & 0x00ff;
    sum += NBITS[temp];
    return(sum);
}

/************************************************************************
/* my_alloc returns the address to a pre-allocated pool of free store */
/************************************************************************

char *
my_alloc(size)
int size;
{
    char *retpt;
    int firstblock;
    if (freestart == NULL) {
        printf("Error in my_alloc: couldn't make first allocate of %d byte
        \n", firstblock);
        exit(0);
    }
    numalloc++;
    freestart = freestart + (sizeof(char)*firstblock) - 1;
    return(retpt);
}
inl_gate_list generates the gate list and the free list with total gate gates.

```c
void inl_gate_list(int total_gate)
{
    G_pter tmpg;
    G_list tmpg;
    gate_head = NULL;
    free_head = NULL;
    for (i=1; i<total_gate; i++) {
        if (gate_head == NULL)
            gate_head = (G_pter)my_alloc(sizeof(struct G_node));
        else
            tmpg->Next = (G_pter)my_alloc(sizeof(struct G_node));
        if (free_head == NULL)
            free_head = (G_list)my_alloc(sizeof(struct G_list));
        else
            tmfg->Next = (G_list)my_alloc(sizeof(struct G_list));
        tmpg->Indx = i;
        tmpg->Owner = NOOWNER;
        tmpg->XPos = NOPOS;
        tmpg->YPos = NOPOS;
        tmpg->Placed = FALSE;
        tmpg->Nets = NULL;
        tmpg->Next = NULL;
        tmpg->Gate = tmpg;
        tmfg->Gate = tmfg;
    }
}
```

inl_net_list generates the net list with total nets nets.

```c
void inl_net_list(int total_nets)
{
    N_pter tmpn;
    nets_head = NULL;
    for (i=1; i<total_nets; i++) {
        if (nets_head == NULL)
            nets_head = (N_pter)my_alloc(sizeof(struct N_node));
        else
            tmpn->Next = (N_pter)my_alloc(sizeof(struct N_node));
        tmpn = tmpn->Next;
    }
}
```
void link_nodes(tmpn, nbers)
N_pter tmpn;
int nbers;
{
    int i, indx;
    Gls_pt tmgl;
    G_pter tmgg;
    Nls_pt tmnl;
    for (i=1; i<=nbers; i++) {
        indx = get_input();
        tmgg = gate_head;
        while (tmgg->Indx != indx) tmgg = tmgg->Next;
        if (tmpp->Gate == NULL) {
            tmpn->Gate = Gls_pt(my_alloc(sizeof(struct G_list)));
            tmgl = tmpn->Gate;
        } else {
            tmgl->Next = Gls_pt(my_alloc(sizeof(struct G_list)));
            tmgl = tmgl->Next;
        }
        tmgl->Gate = tmgg;
        tmgl->Next = NULL;
        tmnl = (Nls_pt)(my_alloc(sizeof(struct N_list)));
        tmnl->Next = tmgg->Nets;
        tmgg->Nets = tmnl;
        tmnl->Nets = tmpn;
    }
}

void readin()
{
    N_pter tmpn;
    total_g = get_input();
    total_n = get_input();
    inl_gate_list(total_g);
    inl_net_list(total_n);
    tmpn = nets_head;
    while (tmpn != NULL) {
        nbers = get_input();
        tmpn
link_nodes(tmpn,nbers);
    tmpn = tmpn->Next;
}

/**************************************************************************/
/* rand returns a random number between zero and max */
int rand(int max)
int max;
    return( (int)((random())/(MAXINT)*(max+1)) );

/**************************************************************************/
/* rem_free removes the designated gate from the free list */
void rem_free(int ind)
int ind;
{
Gls_pt tmfg1=free_head;
Gls_pt tmfg2=free_head->Next;
int found = FALSE;
if (free_head->Gate->Indx == ind) {
    tmfg1 = free_head;
    free_head = tmfg1->Next;
    shfree((Gls_pt) tmfg1);
    total_alloc -= (sizeof(struct G_list));
    found = TRUE;
}
else while (tmfg2 != NULL && found == FALSE) {
    if (tmfg2->Gate->Indx == ind) {
        tmfg1->Next = tmfg2->Next;
        shfree((Gls_pt) tmfg2);
        total_alloc -= (sizeof(struct G_list));
        found = TRUE;
    } else {
        tmfg1 = tmfg2;
        tmfg2 = tmfg2->Next;
    }
}
}

/**************************************************************************/
/* add_free adds the designated gate to the free list */
/* returns false if gate is already on the free list */
int add_free(int ind, Q)
int ind;
Q_pter Q;
{
Gls_pt tmfg=Q->Gatehead;
Gls_pt tmfg2=Q->Gatehead->Next;
G_pter tmpq=gate_head;
int found=FALSE;
int not_already_free=TRUE;

    while (tmfg2 != NULL && not_already_free == TRUE) {

}
if (tmfg2->Gate->Indx != index) not_already_free = FALSE;
else tmfg2 = tmfg2->Next;

if (not_already_free == TRUE) {
    while (tmpg != NULL && found == FALSE) {
        if (tmpg->Indx == index) found = TRUE;
        else tmpg = tmpg->Next;
    }
    if (tmpg == NULL) {
        Q->Gatehead = (Gls_pt)my_alloc(sizeof(struct G_list));
        Q->Gatehead->Next = tmfg;
        Q->Gatehead->Gate = tmfg;
    } else {
        printf("Error in add_free: gate index doesn't exist\n");
        exit(0);
    }
}
return(not_already_free);

/**************************************************************************
// assign gate from free list to quadr (update nets' Quads fields; /*
// update gate's Owner field) if ind.x not in free list then return false
/**************************************************************************/
int assign(indx, quadr)
int ind.x, quadr;
{
    Gls_pt tmfg = free_.head;
    Nls_pt tmnl;
    int found = FALSE;
    int this_bit;
    while (tmfg != NULL && found == FALSE) {
        if (tmfg->Gate->Indx == index) {
            found = TRUE;
            tmfg->Gate->Owner = quadr;
            QP[quadr]++;
            if (QP[quadr] == QPmax) QPmax = QP[quadr];
            tmnl = tmfg->Gate->Nets;
            while (tmnl != NULL) {
                tmnl->Nets->Quads |= (unshort)BIT(quadr);
                tmnl = tmnl->Next;
            }
            rem_free(indx);
        }
        else tmfg = tmfg->Next;
    }
    return(found);
}

/**************************************************************************
// assign_seeds assigns each quadrant a random gate */
/**************************************************************************/
void assign_seeds()
{
    Gls_pt tmgl;
    int row, col;
    int quadr;
    int gate;
    int count;

switch (SEEDMETHOD) {
case ALL:
    for (row=0; row<NROWS; row++)
        for (col=0; col<NCOLS; col++) {
            quadr = (4*row)+col;
            gate = (rand(numgates-1)+1);
            tmsgl = free_head;
            for (count=1; count<gate; count++) tmsgl = tmsgl->Next;
            gate = tmsgl->Gate->Index;
            while (assign(gate,quadr) == FALSE) {
                gate = (rand(numgates-1)+1);
                tmsgl = free_head;
                for (count=1; count<gate; count++) tmsgl = tmsgl->Next;
                gate = tmsgl->Gate->Index;
            }
            break;
        case MIDDLE:
            row = ((NROWS-1)/2);
            col = ((NCOLS-1)/2);
            quadr = (4*row)+col;
            gate = (rand(numgates-1)+1);
            tmsgl = free_head;
            for (count=1; count<gate; count++) tmsgl = tmsgl->Next;
            gate = tmsgl->Gate->Index;
            while (assign(gate,quadr) == FALSE) {
                gate = (rand(numgates-1)+1);
                tmsgl = free_head;
                for (count=1; count<gate; count++) tmsgl = tmsgl->Next;
                gate = tmsgl->Gate->Index;
            }
            break;
        case NONE:
            break;
        default:
            printf("ERROR in assign_seeds: SEEDMETHOD undefined");
            exit(0);
            break;
    }

/*********************************************************************/
/* exponent returns x to the y power */
/*********************************************************************/
int exponent(x,y)
float x;
int y;
{
    int count;
    float temp=1.0;
    if (y > 0) for (count=0; count<y; count++) temp *= x;
    if (y == 0) temp = 0.0;
    if (y < 0) temp = (-1) * exponent(x,(-1)*y);
    return ((int)temp);
}

/*********************************************************************/
/* set_ceiling sets QPmax when the ceiling size balancing is used */
/*********************************************************************/
void set_ceiling(P) Q_seter P;
int i = 1;
QPceil = 1;
while (i < MAXDIV & P->div[i] > 1) {
    QPceil *= (int)((int)P->div[i] * (int)P->div[i]);
    i++;
}

/** For each quadrant, find gate with highest affinity factor */
/***************************************************************************/
void max_affinity()
|
Gls_pt tmfg;
Nls_pt tmnl;
int row, col;
int quadr, nghbd;
double affin, maxaffin;
int maxindx;

while ((quadrant.next() - 1) <= (4*(NROWS-1) + (NCOLS-1)) ) {
    for (row = 0; row < NROWS; row++)
        for (col = 0; col < NCOLS; col++) {
            if (quadrant == ((4*row) + col)) {
                maxaffin = (-1.0)*MAXINT;
                tmfg = free_head;
                while (tmfg != NULL) {
                    switch (BALMETHOD) {
                    case JUSTLIN:
                        affin = (BWF * (((int) (QPmax*SF[quadr])) - QP[quadr]));
                        break;
                    case JUSTEXP:
                        affin = exponent(BWF, (((int) (QPmax*SF[quadr])) - QP[quadr]));
                        break;
                    case CEILLIN:
                        if (QP[quadr] >= ((int) (QPceil*SF[quadr]))) affin = (-1*MAXINT/2);
                        else
                            affin = (BWFCEIL * (((int) (QPceil*SF[quadr])) - QP[quadr]));
                        break;
                    case CEILEXP:
                        if (QP[quadr] >= ((int) (QPceil*SF[quadr]))) affin = (-1*MAXINT/2);
                        else
                            affin = exponent(BWFCEIL, (((int) (QPceil*SF[quadr])) - QP[quadr]));
                        break;
                    default:
                        printf("ERROR in max_affin: BALMETHOD not defined\n");
                        exit(0);
                        break;
                    }
                }
                tmnl = tmfg->Gate->Nets;
                while (tmnl != NULL)
                    for (nghbd = 0; nghbd < 5; nghbd++) {
                        affin += (WF[nghbd]*numbits(MASK[quadr][nghbd]) & (tmnl->Nets->Quads)) ;
                    }
                if (affin > maxaffin) {
                    maxaffin = affin;
                    maxindx = tmfg->Gate->Indx;
                }
            } else
                affin = exponent(BWFCEIL, (((int) (QPceil*SF[quadr])) - QP[quadr]));
                    break;
                default:
                    printf("ERROR in max_affin: BALMETHOD not defined\n");
                    exit(0);
                    break;
                }
            } else
                affin = exponent(BWFCEIL, (((int) (QPceil*SF[quadr])) - QP[quadr]));
                        break;
                    default:
                        printf("ERROR in max_affin: BALMETHOD not defined\n");
                        exit(0);
                        break;
                    }
                tmnl = tmfg->Gate->Nets;
                while (tmnl != NULL)
                    for (nghbd = 0; nghbd < 5; nghbd++) {
                        affin += (WF[nghbd]*numbits(MASK[quadr][nghbd]) & (tmnl->Nets->Quads)) ;
                    }
                if (affin > maxaffin) {
                    maxaffin = affin;
                    maxindx = tmfg->Gate->Indx;
                }
void arbitrate()
{
    int row, col;
    int quadr;
    double maxaffin = (-1.0) * MAXINT;
    int success = FALSE;
    /* first pass */
    for (row = 0; row < NROWS; row++)
        for (col = 0; col < NCOLS; col++)
            quadr = (4 * row) + col;
            if (wish_list[quadr].AFFIN > maxaffin)
                maxaffin = wish_list[quadr].AFFIN;
    /* second pass */
    for (row = 0; row < NROWS; row++)
        for (col = 0; col < NCOLS; col++)
            quadr = (4 * row) + col;
            if (wish_list[quadr].AFFIN == maxaffin)
            {
                success = assign(wish_list[quadr].INDX, quadr);
            }
}

int qrow(quadr)
{
    int quadr;
    return ((int)(quadr / 4));
}

int qcol(quadr)
{
    int quadr;
    return ((int)(quadr % 4));
}

int wire_cost()
N_pter tmpn = nets_head;
Gls_pt tmgl;
int row, col;
int quad;
int max_x, min_x;
int max_y, min_y;
short xpos, ypos;
int cost=0;
int found_one;

while (tmpn != NULL) {
    int max_x = -1;
    int min_x = MAXINT;
    int max_y = -1;
    int min_y = MAXINT;
    tmgl = tmpn->Gate;
    found_one = FALSE;
    while (tmgl != NULL) {
        if (found_one == TRUE) retun;
        found_one = TRUE;
        xpos = tmgl->Gate->XPos;
        ypos = tmgl->Gate->YPos;
        if (xpos > max_x) max_x = xpos;
        if (xpos < min_x) min_x = xpos;
        if (ypos > max_y) max_y = ypos;
        if (ypos < min_y) min_y = ypos;
        tmgl = tmgl->Next;
    }
    if (found_one == TRUE) cost += (max_x-min_x)+(max_y-min_y);
    tmpn = tmpn->Next;
    }
}

return(cost);

/************************************************************•**********/
/* setprocs ensures that NUMPROCS is O.K. and then calls m_set_procs */
int setprocs()
{
    int maxprocs;

    maxprocs = cpus_online()-1;
    if (NUMPROCS > maxprocs) NUMPROCS = maxprocs;
    if (NUMPROCS < 1) NUMPROCS = 1;
    m_set_procs(NUMPROCS);
    return(m_get_numprocs());
}

/******************************************************•**********/
/* output_results prints the results */
void output_results()
{
    fflush(stdout);
    printf("\n");
    printf("\n");
    printf("\n");
    printf("Size of Test Case:
-----------------
Number of gates in input: %d
Number of nets in input: %d\n",total_g);
    printf("\n");
    printf("\n");
}
flush(stdout);
printf("Weighting Factors:\n");
printf("-------------------\n");
printf("Neighborhood #0: %d\n", WF(0));
printf("Neighborhood #1: %d\n", WF(1));
printf("Neighborhood #2: %d\n", WF(2));
printf("Neighborhood #3: %d\n", WF(3));
printf("Neighborhood #4: %d\n", WF(4));
if (BALMETHODOF = CEILLIN || BALMETHODOF = CEILEXP)
printf("Maintain Balance: \$7.5f\n", BWFCEIL);
else printf("Maintain Balance: \$7.5f\n", BWF);
flush(stdout);
}

if (ANNEAL==TRUE) {
printf("Simulated Annealing Parameters:\n");
printf("-----------------------------\n");
printf("Attempts per gate before failure: \$d\n", FAILMULT);
printf("Accepts per gate per temperature: \$d\n", ACCEPTMULT);
printf("Consecutive fails before stopping: \$d\n", CONSECFAIL);
printf("Cool Down multiplier: \%5.2f\n", COOLDOWN);
printf("Heat Up multiplier: \%5.3f\n", HEATUP);
}

flush(stdout);
switch (SEEDMETHOD) {
case ALL:
  printf("Seed Assignment: Assign all quadrants a random seed\n");
  break;
case MIDDLE:
  printf("Seed Assignment: Assign middle quadrant a random seed\n");
  break;
case NONE:
  printf("Seed Assignment: No seeds assigned\n");
  break;
default:
  printf("Seed Assignment: Error none chosen\n");
  break;
}
flush(stdout);
switch (BALMETHODOF) {
case JUSTLIN:
  printf("Balance Method: Linear w/o Ceiling\n");
  break;
case JUSTEXP:
  printf("Balance Method: Exponential w/o Ceiling\n");
  break;
case CEILLIN:
  printf("Balance Method: Linear with Ceiling\n");
  break;
case CEILEXP:
  printf("Balance Method: Exponential with Ceiling\n");
  break;
default:
  printf("Balance Method: Error none chosen\n");
  break;
}
flush(stdout);
switch (GLOBALMETHOD) {
case NOGLOBMETH:
  printf("Global Method: No Awareness of other quadrants\n");
  break;
case ONLYPLACED:
printf("Global Method: Aware of already placed ones only\n");
case ALL_MIDDLE:
printf("Global Method: Aware of all gates - Middle One\n");
case ALL_MIDTWO:
printf("Global Method: Aware of all gates - Middle One or Two\n");
case ALL_WHOLE:
printf("Global Method: Aware of all gates - Whole Row or Col\n");
default:
printf("Global Method: Error: none chosen!!\n");

fclose(stdout);
switch (QORDER) {
case BREADTH:
printf("Partition Order: Breadth-first\n");
case DEPTH:
printf("Partition Order: Depth-first\n");
default:
printf("Partition Order: Error: none chosen\n");
}
fclose(stdout);
switch (DIVORDER) {
case ASCEND:
printf("Size Ordering: Ascending order\n");
case DESCEND:
printf("Size Ordering: Descending order\n");
default:
printf("Size Ordering: Error: none chosen\n");
}
fclose(stdout);
switch (DIVWITH4) {
case TRUE:
printf("16-way partitions: Allowed\n");
case FALSE:
printf("16-way partitions: Not Allowed\n");
default:
printf("16-way partitions: Error: not defined\n");
}
printf("\n");
printf("Assignments of gates to Grid Points:\n");
printf("-----------------------------\n");
printf("\n");
fclose(stdout);
print_placement();
printf("\n");
printf("\n");
printf("Memory Usage:\n");
printf("-----------------------\n");
printf("\n");
printf("\n");
printf("Number of calls to malloc: \d\n","numalloc\n");
printf("Allocated memory: \d bytes\n","total_alloc\n");
printf("Unused memory: \d bytes\n","total_alloc-freeend\n");
printf("\n");
fflush(stdout);
printf("Performance measures:\n");
printf("-----------------------------\n");
printf("Number of processors used: %d, setprocs_ret:\n");
printf("Estimated wiring cost: %d, wire_cost():\n");
printf("Computation time: %5.3f seconds, DeltaT: \n");
printf("\n");
fflush(stdout);

double
	calc_affinity(quadr, indx)
	int quadr;
	int indx;
{
Gls_pt tmfg;
Nls_pt tmnl;
int row, col;
int nghbd;
double affin=MAXINT;

for (row=0; row<ROWS; row++)
for (col=0; col<COLS; col++)
if (quadr == ((4*row)+col)) {
	tmfg = free_head;
while (tmfg != NULL & tmfg->Gate->Ind.x == indx) tmfg = tmfg->Next;
if (tmfg == NULL) {
	printf("Error in calc affinity: indx not a free gate\n");
exit(0);
}
switch (BALMETH)
{
case JUSTLIN:
	affin = (BWF * ((int) (QPmax*SF[quadr])) -QP[quadr]));
	break;
case JUSTEXP:
	affin = exponent (BWF, ((int) (QPmax*SF[quadr])) -QP[quadr]));
	break;
case CEILLIN:
	if (QP[quadr] >= ((int) (QPceil*SF[quadr])))
		affin = (-1* (MAXINT/2));
	else
		affin = (BWFCEIL * ((int) (QPceil*SF[quadr])) -QP[quad]));
	break;
case CEILEXP:
	if (QP[quadr] >= ((int) (QPceil*SF[quadr])))
		affin = (-1* (MAXINT/2));
	else
		affin = exponent (BWFCEIL, ((int) (QPceil*SF[quadr])) -QP[quad]));
	break;
default:
	printf("ERROR in calc affinity: BALMETH not defined\n");
exit(0);
break;
}
tmnl = tmfg->Gate->Nets;
while (tmnl != NULL) {
for (nghbd=0; nghbd<5; nghbd++)
	affin += (WF[nghbd]*numbits (MASK[quadr][nghbd] & (tmnl->Nets->Quads)));
}
tmnl = tmnl->Next;
}
if (affin == MAXINT) {
    printf("Error in calc_affinity: illegal quadrant number\n");
    exit (0);
} else {
    return (affin);
}

/************************************************************
/* round_up takes a double and rounds it up to an integer */
int round_up(x)
    double x;
{
    int retval;
    retval = (int)x;
    if (x > (double)retval) retval++;
    return (retval);
}

/************************************************************************
* init_Q creates the head of the Q tree *
************************************************************************/
void init_Q()
{
    int n = 0;    /* number of twos */
    int m = 0;    /* number of threes */
    int p = 0;    /* number of fours */
    int one = 1;
    int maxm;
    int s;
    int tprod;
    int min;
    int nmin, nmin;
    int i;
    int row, col;
    float size_mult;
    int temp:

    /*************************************************************************
    /* Calculate p, m, and n */
    /*************************************************************************
    tprod = 1;
    maxm = 0;
    s = round_up(sqrt((double)total_g)); /* Determine s */
    while (tprod < s) {
        tprod *= 3;
        maxm++;
    }
    min = MAXINT; /* Find all possible factors*/
    for (m=0; m<maxm; m++) {
        n = 0;
        tprod = (int)pow(3.0, (double)m);
        while (tprod < s) {
            tprod *= 2;
            n++;
        }
        if (tprod < min) {
            min = tprod;
            nmin = n;
            nmin = m;
        }
    }
}
n = n_min;
m = m_min;
if (DIVWITH4 == TRUE) {
    p = (n/2);
n = (n%2);
} else p = 0;
length_x = s;
length_y = s;

/**********************************************************/
/* Initialize list Q node*/
/**********************************************************/
Q_head = (Q_ptr)malloc(sizeof(struct Q_node));
Q_head->type = CORNER;
Q_head->xlo = 0;
Q_head->ylo = 0;
Q_head->xhi = min-1;
Q_head->yhi = min-1;
switch (DIVORDER) {
case ASCEND:
    for (i=0; i<MAXDIV; i++) {
        if (n>0) {
            Q_head->div[i] = 2;
n--;
        } else if (m>0) {
            Q_head->div[i] = 3;
m--;
        } else if (p>0) {
            Q_head->div[i] = 4;
p--;
        } else if (one>0) {
            Q_head->div[i] = 1;
one--;
        } else Q_head->div[i] = 0;
        break;
    case DESCEND:
        for (i=0; i<MAXDIV; i++) {
            if (p>0) {
                Q_head->div[i] = 4;
p--;
            } else if (m>0) {
                Q_head->div[i] = 3;
m--;
            } else if (n>0) {
                Q_head->div[i] = 2;
n--;
            } else if (one>0) {
                Q_head->div[i] = 1;
one--;
            } else Q_head->div[i] = 0;
        break;
    default:
        printf("Error in init_Q: DIVWITH4 not defined!\n");
exit(0); break;

Q_head->nrows = Q_head->div[0];
Q_head->ncols = Q_head->div[0];
Q_head->numgates = total_g;
temp = min/Q_head->div[0];
size_mult = (double)(temp-(min-s))/(double)temp;
for (row=0; row<Q_head->nrows; row++) {
    for (col=0; col<Q_head->ncols; col++) {
        if (row == Q_head->nrows-1 && col == Q_head->ncols-1) {
            if (row == Q_head->nrows-1 && col == Q_head->ncols-1) {
                Q_head->sf[(4*row)+col] = size_mult * size_mult;
            } else Q_head->sf[(4*row)+col] = size_mult;
        } else Q_head->sf[(4*row)+col] = 1.0;
    }
}
Q_head->Gatehead = free_head;
Q_head->Next = NULL;
Q_tail = Q_head;

/**********************************************************************
 * unassign_all clears out Owner, Quad Fields and QP variables  
 * ********************************************************************/

void unassign_all() {
    G_piter tmpg;
    N_piter tmpn;
    int row, col;
    tmpg = gate_head; /* Clear Owner Fields */
    while (tmpg != NULL) {
        tmpg->Owner = NOOWNER;
        tmpg = tmpg->Next;
    }
    tmpn = nets_head; /* Clear Quads Fields */
    while (tmpn != NULL) {
        tmpn->Quads = (unshort)0;
        tmpn = tmpn->Next;
    }
    for (row=0; row<4; row++)
        for (col=0; col<4; col++)
            QP[(4*row)+col] = 0;
}

/**************************************************************************
 * assign_params assigns the values in Q_node pointed to by P and sets  
 * the global partitioning varaibles to these values  
 * ************************************************************************/

void assign_params(P)
Q_piter P;
{
    int row, col;
    NROWS = P->nrows;
    NCOLS = P->ncols;
    numgates = P->numgates;
    for (row=0; row<NROWS; row++)
for (col=0; col<NCOLS; col++)
    SF[(4*row)+col] = P->sf[(4*row)+col];
free_head = P->Gatehead;

/************************************************************************
*/
void set_quads(P)
Q_pter P;
    Gls_pt tmgl;
    Nls_pt tmnl;
    G_pter tmpg;
    short xzone;
    short yzone;
    short zone;
    int r, c;
    int lorow, locol;
    int hirow, hicol;
    int quadr;
    short NOROWCOL = FALSE;
    tmgl2 = P->Gatehead;
    while (tmgl2 != NULL) {
        tmnl = tmgl2->Gate->Nets;
        while (tmnl != NULL) {
            tmgl = tmnl->Gate;
            while (tmgl != NULL) {
                if (tmgl->Gate->XPos != NOPOS && tmgl->Gate->YPos != NOPOS) {
                    if (tmgl->Gate->XPos < P->xlo) xzone = 0;
                        else if (tmgl->Gate->XPos < P->xhi) xzone = 1;
                            else xzone = 2;
                                if (tmgl->Gate->YPos < P->ylo) yzone = 0;
                                    else if (tmgl->Gate->YPos < P->yhi) yzone = 1;
                                        else yzone = 2;
                                            zone = (3*yzone) + xzone;
                                                NOROWCOL = FALSE; /* ZONES: */
                                                switch(zone) {
                                                    case 0:
                                                        lorow = 0;
                                                            /*************/
                                                        hirow = 0;
                                                            /*************/
                                                        locol = 0;
                                                            /*************/
                                                        hicol = 0;
                                                            /*************/
                                                        break;
                                                    /*************/
                                                    case 1:
                                                        lorow = 0;
                                                            /*************/
                                                        hirow = 0;
                                                            /*************/
                                                        locol = 1;
                                                            /*************/
                                                        hicol = (double)(tmgl->Gate->XPos - P->xlo) / (double)(P->xhi - P->xlo + 1)) * (double)P->ncols;
                                                            hicol = locol;
                                                            /*************/
                                                        else { switch (GLOBALMETHOD) {
                                                            case ALL_MIDTWO:
                                                                hicol = locol; locol = hicol;
                                                                        break;
                                                            case ALL_MIDTWO:
                                                                hicol = locol; locol = hicol;
                                                            /* ZONES: */
                                                            switch (zone) {
                                                                case 0:
                                                                    lorow = 0;
                                                                        /*************/
                                                                    hirow = 0;
                                                                        /*************/
                                                                    locol = 0;
                                                                        /*************/
                                                                    hicol = 0;
                                                                        /*************/
                                                                    break;
                                                                /*************/
                                                                case 1:
                                                                    lorow = 0;
                                                                        /*************/
                                                                    hirow = 0;
                                                                        /*************/
                                                                    locol = 1;
                                                                        /*************/
                                                                    hicol = (double)(tmgl->Gate->XPos - P->xlo) / (double)(P->xhi - P->xlo + 1)) * (double)P->ncols;
                                                                        hicol = locol;
                                                                    /*************/
                                                                else { switch (GLOBALMETHOD) {
                                                                    case ALL_MIDTWO:
                                                                        hicol = locol; locol = hicol;
                                                                    /* ZONES: */
                                                                    switch (zone) {
                                                                        case 0:
                                                                            lorow = 0;
                                                                            /*************/
                                                                            hirow = 0;
                                                                            /*************/
                                                                            locol = 0;
                                                                            /*************/
                                                                            hicol = 0;
                                                                            /*************/
                                                                            break;
                                                                        /* ZONES: */
                                                                        switch (zone) {
                                                                            case 0:
                                                                                lorow = 0;
                                                                                /*************/
                                                                                hirow = 0;
                                                                                /*************/
                                                                                locol = 0;
                                                                                /*************/
                                                                                hicol = 0;
                                                                                /*************/
                                                                                break;
                                                                            /*************/
                                                                            case 1:
                                                                                lorow = 0;
                                                                                /*************/
                                                                                hirow = 0;
                                                                                /*************/
                                                                                locol = 1;
                                                                                /*************/
                                                                                hicol = (double)(tmgl->Gate->XPos - P->xlo) / (double)(P->xhi - P->xlo + 1)) * (double)P->ncols;
                                                                                hicol = locol;
                                                                            /*************/
                                                                        }
if ((P->ncols % 2) == 0)
    locol = hicol = 1;
else locol = hicol;
break;
case ALL_WHOLE:
    locol = 0;
    hicol = (P->ncols - 1);
    break;
default:
    printf("Error in set_quads: GLOBALMETHOD has illegal value
of %d",GLOBALMETHOD);
    break;
}
locol = 0;
hicol = 0;
break;
case 2:
    lorow = 0;
    hirow = 0;
    locol = (P->ncols - 1);
    hicol = locol;
    break;
case 3:
    if (tmgl->Gate->Placed == TRUE) {
        lorow = ( ((double) (tmgl->Gate->YPos - P->ylo) / (double) (P->yhi - P->ylo + 1)) * (double) P->nrows);
        hirow = lorow;
    } else {
        switch (GLOBALMETHOD) {
        case ALL_MIDDLE:
            hirow = P->nrows / 2;
            lorow = hirow;
            break;
        case ALL_MIDTWO:
            hirow = P->nrows / 2;
            if (((P->nrows % 2) == 0))
                lorow = hirow - 1;
            else lorow = hirow;
            break;
        case ALL_WHOLE:
            lorow = 0;
            hirow = (P->ncols - 1);
            break;
        default:
            printf("Error in set_quads: GLOBALMETHOD has illegal value
of %d",GLOBALMETHOD);
            break;
        }
        lorow = 0;
        hicol = 0;
        break;
    }
case 4:
    NOROWCOL = TRUE;
    break;
case 5:
    if (tmgl->Gate->Placed == TRUE) {
        lorow = ( ((double) (tmgl->Gate->YPos - P->ylo) / (double) (P->yhi - P->ylo + 1)) * (double) P->nrows);
        hirow = lorow;
    } else {
switch (GLOBALMETHOD) {
    case ALL_MIDDLE:
        hirow = P->nrows / 2;
        lorow = hirow;
        break;
    case ALL_MIDTWO:
        hirow = P->nrows / 2;
        if ((P->nrows % 2) == 0)
            lorow = hirow - 1;
        else
            lorow = hirow;
        break;
    case ALL_HALF:
        lorow = 0;
        hirow = (P->ncols - 1);
        break;
    default:
        printf("Error in set_quads: GLOBALMETHOD has illegal value of %d",GLOBALMETHOD);
        break;
    }
locol = (P->ncols - 1);
    hicol = locol;
    break;
    case 6:
        lorow = (P->nrows - 1);
        hirow = lorow;
        locol = 0;
        hicol = 0;
        break;
    case 7:
        lorow = (P->nrows - 1);
        hirow = lorow;
        if (tmgl->Gate->Placed == TRUE) {
            locol = ((double)(tmgl->Gate->XPos - P->xlo) / (double)(P->xhi - P->xlo + 1)) * (double)P->ncols;
            hicol = locol;
        } else {
            switch (GLOBALMETHOD) {
                case ALL_MIDDLE:
                    hicol = P->ncols / 2;
                    locol = hicol;
                    break;
                case ALL_MIDTWO:
                    hicol = P->ncols / 2;
                    if ((P->ncols % 2) == 0)
                        locol = hicol - 1;
                    else
                        locol = hicol;
                    break;
                case ALL_WHOLE:
                    locol = 0;
                    hicol = (P->ncols - 1);
                    break;
                default:
                    printf("Error in set_quads: GLOBALMETHOD has illegal value of %d",GLOBALMETHOD);
                    break;
            }
        }
        break;
    case 8:
        lorow = (P->nrows - 1);
        hirow = lorow;
        if (P->nrows % 2 == 0) {
            locol = (((double)(P->nrows + 1) - lorow) / (double)(P->nrows - 1)) * (double)P->ncols;
            hicol = locol;
        } else {
            switch (GLOBALMETHOD) {
                case ALL_MIDDLE:
                    hicol = P->nrows / 2;
                    locol = hicol;
                    break;
                case ALL_MIDTWO:
                    hicol = P->nrows / 2;
                    if ((P->nrows % 2) == 0)
                        locol = hicol - 1;
                    else
                        locol = hicol;
                    break;
                case ALL_HALF:
                    locol = 0;
                    hicol = (P->nrows - 1);
                    break;
                default:
                    printf("Error in set_quads: GLOBALMETHOD has illegal value of %d",GLOBALMETHOD);
                    break;
            }
        }
        break;
    case 9:
        lorow = (P->nrows - 1);
        hirow = lorow;
        if (P->nrows % 2 == 0) {
            locol = (((double)(P->nrows + 1) - lorow) / (double)(P->nrows - 1)) * (double)P->ncols;
            hicol = locol;
        } else {
            switch (GLOBALMETHOD) {
                case ALL_MIDDLE:
                    hicol = P->nrows / 2;
                    locol = hicol;
                    break;
                case ALL_MIDTWO:
                    hicol = P->nrows / 2;
                    if ((P->nrows % 2) == 0)
                        locol = hicol - 1;
                    else
                        locol = hicol;
                    break;
                case ALL_HALF:
                    locol = 0;
                    hicol = (P->nrows - 1);
                    break;
                default:
                    printf("Error in set_quads: GLOBALMETHOD has illegal value of %d",GLOBALMETHOD);
                    break;
            }
        }
        break;
    case 10:
        lorow = (P->nrows - 1);
        hirow = lorow;
        if (P->nrows % 2 == 0) {
            locol = (((double)(P->nrows + 1) - lorow) / (double)(P->nrows - 1)) * (double)P->ncols;
            hicol = locol;
        } else {
            switch (GLOBALMETHOD) {
                case ALL_MIDDLE:
                    hicol = P->nrows / 2;
                    locol = hicol;
                    break;
                case ALL_MIDTWO:
                    hicol = P->nrows / 2;
                    if ((P->nrows % 2) == 0)
                        locol = hicol - 1;
                    else
                        locol = hicol;
                    break;
                case ALL_HALF:
                    locol = 0;
                    hicol = (P->nrows - 1);
                    break;
                default:
                    printf("Error in set_quads: GLOBALMETHOD has illegal value of %d",GLOBALMETHOD);
                    break;
            }
        }
        break;
}
hirow = lorow;
locol = (P->ncols - 1);
hicol = locol;
break;

default:
    printf("Error in set_quads: unacceptable zone\n");
    exit(0);
    break;
}
if (NOROWCOL == FALSE) {
    for (r=lorow; r<hirow; r++)
        for (c=locol; c<hicol; c++) {
            quadr = (4*r) + c;
            tmnl->Nets->Quads |= (unshort)BIT[quadr];
        }
}
tmgl = tmgl->Next;
    .
tmnl = tmnl->Next;
    .
tmgl2 = tmgl2->Next;
}

*******************************************************************************
/* placement takes a pointer into the Q tree and tries to make more children */
/* if it's possible (returns TRUE if successful) */
*******************************************************************************
int placement(P)
Q_pter P;
{
    Q_pter C;
    Q_pter tmpq;
    Q_pter tmpg;
    Gls_pt tmgl;
    int row, col;
    int quadr;
    int r, c;
    int q;
    int l;
    double size_mult;
    int x,y;
    if (P->div[1] > 1) {
        /************** /
        / Do Partitioning /
        /*******************/
        assign_params(P);
        if (BALMETHOD == CEILLIN || BALMETHOD == CEILEXP) set_ceiling(P);
        unassign_all();
        if (FIRST_TIME == FALSE && GLOBALMETHOD != NOGLOBALMETH) set_quads(P);
        if (FIRST_TIME == TRUE) assign_seeds();
        while (free_head != NULL) {
            m_fork(max_affinity);
            arbitrate();
        }
        /************** /
        / Create Children /
        /*******************/
for (row=0; row<P->nrows; row++)
for (col=0; col<P->ncols; col++) {
    quadr = (4*row)+col;
    switch(QORDER) {
    case BREADTH:
        C = (Q_pter)my_alloc(sizeof(struct Q_node));
        Q_tail->Next = C;
        Q_tail = C;
        C->Next = NULL;
        break;
    case DEPTH:
        C = (Q_pter)my_alloc(sizeof(struct Q_node));
        tmpq = P->Next;
        P->Next = C;
        C->Next = tmpq;
        break;
    default:
        printf("Error in placement: QORDER not defined!\n")
        exit(0);
        break;
    } 
    C->xlo = ((P->xhi- P->xlo +1)/P->div[0]+col)+P->xlo;
    C->xhi = ((P->xhi- P->xlo +1)/P->div[0]+col)+P->xlo +1;
    C->ylo = ((P->yhi- P->ylo +1)/P->div[0]+row)+P->ylo;
    C->yhi = ((P->yhi- P->ylo +1)/P->div[0]+row)+P->ylo +1;
    for (i=0; i< MAXDIV-1; i++)
        C->div[i] = P->div[i+1];
    C->div[MAXDIV-1] = 0;
    /* Give child it's gates */
    C->numgates = 0;
    C->Gatehead = NULL;
    tmpg = gate_head;
    while (tmpg != NULL) {
        if (tmpg->Owner == quad) {
            C->numgates++;
            tmp1 = C->Gatehead;
            C->Gatehead = (G_list_pter)my_alloc(sizeof(struct G_list));
            if (GLOBALMETHOD != NOGLOBMETH && GLOBALMETHOD != ONLYPLACE)
                
            tmpg->XPos = (C->xlo + C->xhi)/2;
            tmpg->YPos = (C->ylo + C->yhi)/2;
        } 
        C->Gatehead->Gate = tmpg;
        C->Gatehead->Next = tmp1;
        tmpg = tmpg->Next;
    }
    /* Corner Piece */
    if (row==P->nrows-1 && col==P->ncols-1 && P->type==CORNER) {
        C->nrows = round_up(sqrt(P->sf(quadr))*(float)P->div[1]);
        C->ncols = C->nrows;
        size_mult = 1.0-(float)C->ncols-(sqrt(P->sf(quadr))*(float)P->div[1]));
    for (r=0; r<C->nrows; r++)
        for (c=0; c<C->ncols; c++) {
            q = (4*r)+c;
            if (c==C->ncols-1 || r==C->nrows-1) {
                C->sf[q] = size_mult;
                if (c==C->ncols-1 && r==C->nrows-1)
C->sf[q] = size_mult;
else C->sf[q] = 1.0;
}
C->type = CORNER;

/****************/
/* Bottom Piece */
/****************/
else if (row == P->nrows - 1 && (P->type == CORNER || P->type == BOTTOM)) {
C->nrows = round_up(P->sf[quadr] + (float)P->div[1]);
C->ncols = P->div[1];
size_mult = 1.0 - (float)C->nrows / (float)P->div[1];
for (r = 0; r < C->nrows; r++) {
    for (c = 0; c < C->ncols; c++) {
        q = (4*r) + c;
        if (r == C->nrows - 1) C->sf[q] = size_mult;
        else C->sf[q] = 1.0;
    }
}
C->type = BOTTOM;

/****************/
/* Side Piece */
/****************/
else if (col == P->ncols - 1 && (P->type == CORNER || P->type == SIDE)) {
C->nrows = P->div[1];
C->ncols = round_up(P->sf[quadr] + (float)P->div[1]);
size_mult = 1.0 - (float)C->ncols / (float)P->div[1];
for (r = 0; r < C->nrows; r++) {
    for (c = 0; c < C->ncols; c++) {
        q = (4*r) + c;
        if (c == C->ncols - 1) C->sf[q] = size_mult;
        else C->sf[q] = 1.0;
    }
}
C->type = SIDE;

/****************/
/* Center Piece */
/****************/
else {
    C->nrows = P->div[1];
    C->ncols = P->div[1];
    for (r = 0; r < C->nrows; r++) {
        for (c = 0; c < C->ncols; c++) {
            q = (4*r) + c;
            C->sf[q] = 1.0;
        }
    }
}
C->type = CENTER;
}

return(TRUE);

else if (P->div[1] == 1) {
    /*************************************************************************/
    /* Do Partitioning */
    /*************************************************************************/
assign_params(P);
if (BALMETH == CEILLIN || BALMETH == CEILEXP) set_ceiling(P);
unassign_all();
if (FIRST_TIME == FALSE && GLOBALMETHOD != NOGLOBMETH) set_quads(P);
if (FIRST_TIME == TRUE) assign_seeds();
while (free_head != NULL) {
  m_fork(max_affinity);
  arbitrate();
}

/******************/
/* Assign Position */
/******************/

tmpg = gate_head;
while (tmpg != NULL) {
  if (tmpg->Owner != NOOWNER) {
    x = P->xlo + qcol(tmpg->Owner);
    y = P->ylo + qrow(tmpg->Owner);
    gate_at[x][y] = tmpg;
    tmpg->XPos = x;
    tmpg->YPos = y;
    tmpg->Placed = TRUE;
  }
  tmpg = tmpg->Next;
}
return(FALSE);

/**************************************************************************/
/* print_placement prints out the placement assignments from gate_at */
/**************************************************************************/

void print_placement() {
  int x, y;
  int gates = 0;
  printf(" ");
  for (x=0; x<length_x; x++) printf("%4d",x);
  printf("\n");
  printf(" ");
  for (x=0; x<length_x; x++) printf("------");
  printf("\n");
  for (y=0; y<length_y; y++) {
    printf("%4d",y);
    for (x=0; x<length_x; x++) {
      if (gate_at[x][y] != NULL) {
        printf("%4d",gate_at[x][y]->Index);
        gates++;
      }
      else printf(" ");
    }
    printf("\n");
  }
  if (gates != total_g) printf("Error found in print_placement: gates = %d, total_g = %d\n", gates, total_g);
}

/***************************************************************************/
/* part_wire_cost determines the wire cost for the nets that are */
/* connected to the designated gates */
/***************************************************************************/
double 
part_wire_cost(tmpgl,tmpg2)
G_pter tmpgl;
G_pter tmpg2;
{
    double cost=0;
    Nls_pt tmnl;
    Nl_pt tmpn;
    Gls_pt tmpgl;
    int max_x, min_x;
    int max_y, min_y;
    int xpos, ypos;
    int found_one;
    int net_between;

    /***************************************************************************/
    /* Find wiring cost for tmpgl */
    /***************************************************************************/
    if (tmpgl != NULL) tmnl = tmpgl->Nets;
    else tmnl = NULL;
    while (tmnl != NULL) {
        tmpn = tmnl->Nets;
        max_x = -1;
        min_x = MAXINT;
        max_y = -1;
        min_y = MAXINT;
        tmgl = tmpn->Gate;
        found_one = FALSE;
        while (tmgl != NULL) {
            found_one = TRUE;
            xpos = tmgl->Gate->XPos;
            ypos = tmgl->Gate->YPos;
            if (xpos > max_x) max_x = xpos;
            if (xpos < min_x) min_x = xpos;
            if (ypos > max_y) max_y = ypos;
            if (ypos < min_y) min_y = ypos;
            tmgl = tmgl->Next;
        }
        if (found_one == TRUE)
            cost += (double)((max_x-min_x)*(max_y-min_y));
        tmnl = tmnl->Next;
    }

    /***************************************************************************/
    /* Find wiring cost for tmpg2 */
    /***************************************************************************/
    if (tmpg2 != NULL) tmnl = tmpg2->Nets;
    else tmnl = NULL;
    while (tmnl != NULL) {
        tmpn = tmnl->Nets;
        max_x = -1;
        min_x = MAXINT;
        max_y = -1;
        min_y = MAXINT;
        tmgl = tmpn->Gate;
        found_one = FALSE;
        net_between = FALSE;
        while (tmgl != NULL) {
            if (tmgl->Gate != tmpgl) /*don't count connections twice*/
                found_one = TRUE;
            xpos = tmgl->Gate->XPos;
            ypos = tmgl->Gate->YPos;
            if (xpos > max_x) max_x = xpos;
            if (xpos < min_x) min_x = xpos;
            if (ypos > max_y) max_y = ypos;
            if (ypos < min_y) min_y = ypos;
        }
    }
else net_between = TRUE; /* net connects tmpgl and tmpg2 */
    tnml = tmml->Next;
}
if (found_one = TRUE && net_between == FALSE)
    cost = (double)((max - min_x)*(max - min_y));
    tnml = tmml->Next;
return(cost);

/************************************************************************/
/* sim annealing performs simulated annealing on current assignments */
/*************************************************************************
void
sim_annealing()
{
    G_p ter tmpgl;
    G_p ter tmpg2;
    int x1, y1;
    int x2, y2;
    double temp;
    double cost_before;
    double delta_cost;
    int accepts = 0;
    int bad_accepts = 0;
    int attempts = 0;
    int failures = 0;
    int melted = FALSE;
    int failed = FALSE;
    int next_temp = FALSE;

    printf("Starting Simulated Annealing: ");
    printf("---------------------");

    printf(" Time Temp. Wire Cost Accepts Failure ");

    temp = 1;
    failures = 0;
    while (failures < CONSECFAIL) {
        failed = FALSE;
        while (failed == FALSE) {
            accepts = 0;
            bad_accepts = 0;
            attempts = 0;
            next_temp = FALSE;
            while (next_temp == FALSE && failed == FALSE) {
                printf(" ");
                printf(" ");
                printf(" ");

                x1 = rand(length_x-1);
                x2 = rand(length_x-1);
                y1 = rand(length_y-1);
                y2 = rand(length_y-1);
                tmpgl = gate_at[x1][y1];
tmpg2 = gate_at[x2][y2];
cost_before = part_wire_cost(tmpgl,tmpg2);

/******************************************************************************/
/* Swap the two positions */
/******************************************************************************/
if (tmpgl !• NULL) {
    tmpgl->XPos = x2;
    tmpgl->YPos = y2;
}
if (tmpg2 !• NULL) {
    tmpg2->XPos = x1;
    tmpg2->YPos = y1;
}
gate_at[x1][y1] = tmpg2;
gate_at[x2][y2] = tmpgl;

/******************************************************************************/
/* Decide whether to keep move*/
/******************************************************************************/
delta_cost = part_wire_cost(tmpgl,tmpg2) - cost_before;
if (exp((-1.0 * delta_cost)/temp)>((double)random()/(double)MAXINT)) {
    accepts++;
    if (delta_cost > 0.0) bad_accepts++;
    attempts++;
    if (accepts > (ACCEPTMULT*total_g)) {
        next_temp = TRUE;
        melted = TRUE;
    }
    else if (attempts > (FAILMULT*total_g)) failed = TRUE;
    else {
        /* swap them back again */
        if (tmpgl !• NULL) {
            tmpgl->XPos = x1;
            tmpgl->YPos = y1;
        }
        if (tmpg2 !• NULL) {
            tmpg2->XPos = x2;
            tmpg2->YPos = y2;
        }
        gate_at[x1][y1] = tmpgl;
        gate_at[x2][y2] = tmpg2;
        attempts++;
        if (melted == FALSE) next_temp = TRUE;
        if (attempts > (FAILMULT*total_g)) failed = TRUE;
    }
}
if (next_temp == TRUE) failures=0;
if (melted == TRUE) temp = temp*((double)COOLDOWN);
else temp = temp * HEATUP;
DeltaT = timeDelta(&BeginTime);
printf("%10.2f%10d%12d%12d",DeltaT,temp,wire_cost(),bad_accepts,attempts
-accepts);
fflush(stdout);
if (failed == TRUE) printf(" FAIL");
if (melted == FALSE) printf(" Melting");
printf("\n");
}
failures++;

/******************************************************************************/
/* Random Placement Routine used to test sim_annealing by itself */
/******************************************************************************/
/*************************************************************************/
void
rand_place()
{
    G_ptr tmpg = gate_head;
    int x = 0;
    int y = 0;
    int s;
    s = round_up(sqrt((double)total_g));
    length_x = s;
    length_y = s;
    for (y = 0; y < s; y++) {
        for (x = 0; x < s; x++) {
            if (tmpg != NULL) {
                gate_at[x][y] = tmpg;
                tmpg->XPos = x;
                tmpg->YPos = y;
                tmpg = tmpg->Next;
            }
        }
    }
}

/*****************************************************************************/
/* MAIN */

main()
{
    Q_ptr tmpq;
    printf("\n");
    setprocs_ret = setprocs();
    readin();
    switch (ANNEAL) {
    case FALSE:
        FIRST_TIME = TRUE;
        timeStart(&BeginTime);
        init_Q();
        while (Q_head != NULL) {
            placement(Q_head);
            tmpq = Q_head;
            Q_head = Q_head->Next;
            FIRST_TIME = FALSE;
        }
        DeltaT = timeDelta(&BeginTime);
        m_kill_procs();
        output_results();
        fflush(stdout);
        break;
    case TRUE:
        timeStart(&BeginTime);
        rand_place();
        output_results();
        fflush(stdout);
        timeStart(&BeginTime);
        sim_annealing();
    }
DeltaT = timeDelta(&BeginTime);
output_results();
flush(stdout);
break;

default:
    printf("Error: ANNEAL not defined\n");
    break;