To the Graduate Council:

I am submitting herewith a thesis written by Erica Dionne Wilson entitled "Parallel Algorithms for Corner Stitching." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

D P Mehta

Dr. Dinesh Mehta, Major Professor

We have read this thesis
and recommend its acceptance:

Alfonso Pujol

Ben White

Accepted for the Council:

Carolyn Minkel

Associate Vice Chancellor and
Dean of the Graduate School

Parallel Algorithms for Corner Stitching

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Erica Dionne Wilson

May 1996

Dedication

This thesis is dedicated to my parents

Mr. and Mrs. Herman Wilson

whose unceasing love, support and encouragement have

given me the motivation to excel.

Acknowledgments

I would like to extend special thanks to Dr. Dinesh Mehta, my thesis advisor and chairman whose support and guidance have been very instrumental in the completion of this thesis. Special thanks are also due to my other committee members: Dr. Al Pujol and Dr. Bruce Whitehead for their valuable comments and suggestions.

Other people I would like to thank are the GEM Consortium and the University of Tennessee for providing financial support for my graduate studies, my entire family for being a constant source of love and encouragement and most importantly, God, from whom all good things originate.

Abstract

Corner stitching is a data structuring technique that can be used to represent rectangular objects in interactive VLSI layout editing systems. This thesis proposes parallel algorithms for the batch insertion and deletion operations of the corner stitching data structure. By adding parallel constructs to the serial corner stitching algorithms, new ones are developed that generate multiple streams of instructions and/or data to be executed in parallel on multiple processors. These parallel algorithms were implemented in C on a distributed network composed mainly of SUN workstations using PVM. When efficiently developed, significant run time improvement over the serial version of the batch insertion and deletion operations was observed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The complexity of VLSI (Very Large Scale Integration) circuits has driven designers to rely almost exclusively on computer-aided layout editing tools to generate, modify, and store circuit designs. Integrated circuits are often represented by literally millions of components, so the data structures supporting these layout editing tools must be capable of storing and manipulating a massive amount of information. Because it is sometimes desirable for these editors to be capable of operating in interactive mode, the data structures must also provide data manipulation with a reasonable amount of speed. In particular, it is important that the data structure for a VLSI layout editing system efficiently implements a fairly complex set of design operations and allows fast modification to the database of components stored in the system.

Corner stitching is one such data structuring technique developed by John

Ousterhout [7] to represent rectangular tiles [1] in interactive VLSI layout editing systems. [7] describes three other data structures that are used in VLSI layout editors. These data structuring techniques are linked-list based, bin-based, and neighbor pointers. Although all of these methods have strengths and drawbacks, when compared, corner stitching is the most efficient for use in VLSI layout editing systems. [4] compares corner stitching to two additional data structures: K-D trees and quad trees. Again, corner stitching proved to be the most advantageous method for storing and managing rectangular tiles.

The advantages of corner stitching are a result of two important features: the empty space of circuit layouts is explicitly stored and tiles in the layout are linked together by pointers in a way that allows VLSI CAD operations to be based chiefly on nearby information in the layout. As a result, corner stitching supports fast execution of some basic VLSI CAD operations like neighbor and area searching. It also efficiently supports fast execution of more complex VLSI CAD operations such as plowing and compaction. In addition, the characteristics of corner stitching make fast and easy updating of the database possible.

Despite its advantages over other data structures for representing rectangular tiles, corner stitching has imperfections. One of its shortcomings is that the explicit storage of empty space requires additional memory. Still another area of limitation is with the *point find* operation which returns the location of the tile

[1] A tile refers to any rectangular VLSI component stored in the circuit layout

containing a given point. While most of the VLSI CAD operations supported by corner stitching have average run-times which are linear in the number of tiles near the tile being manipulated, the point find operation has average run-times proportional to the square root of the total number of tiles in the layout. So, if there are millions of tiles in a layout, a few thousands of those could be traversed when performing a point find operation. This is relatively high compared to other data structures such as K-D [6] and quad trees [8] which take $O(\log N)$ time for the point find operation where N is the number of tiles in the layout.

Two of the most used VLSI CAD operations, *tile insertion* and *tile deletion*, make function calls to the point find operation. As a result, the performance of these operations is effected by the $O(\sqrt{N})$ execution time of the point find operation. The effect of the point find operation can be clearly seen when performing the batch variation of these operations. In the batch insertion and deletion operations, a group of tiles are inserted (deleted) into (from) the VLSI layout by reading each tile entry from a file. If a file contains a list of 50,000 tiles to be inserted, the $O(\sqrt{N})$ time for a point find operation will be incurred at least 50,000 times. So, the performance of the batch insertion and deletion operations is an area of the VLSI layout editing problem that can benefit from improvement.

One option for improving the batch insertion and deletion operations is to implement them by applying *parallel processing* principles and techniques. The notion of multiple processors working in parallel to solve problems is not a new

concept. [9] presents a history of research in parallel processing dating back as far as the 1920's. But, over the past decade, research in this area has exploded. One compelling reason for this explosion is outlined in [2] which attributes it to advances in VLSI technology which have made it possible to make very fast processors. According to [2], the speed of today's microprocessors is within one order of magnitude of the speed of the fastest serial processors. This means, if we assemble a number of microprocessors to form a parallel computer, we can obtain raw computing power that overcomes and in many cases, far exceeds that of the fastest serial processor. The performance offered by parallel computers makes the development of parallel algorithms that use these additional resources a viable option for speeding up software applications.

The purpose of this thesis is to develop parallel algorithms for the batch insertion and deletion operations of the corner stitching data structure. Several factors, such as the technique used to partition the work among multiple processors, determine the effectiveness of parallel algorithms and whether or not they improve the performance of a problem. Therefore, three different parallel approaches for improving the performance of the batch insertion and deletion operations will be presented. They are the *Synchronous-Parallel* method, the *Independent-Parallel* method and the *Master/Worker-Parallel* method. The performance of each parallel algorithm will be compared to the serial version of the corner stitching operation in an effort to assess its effectiveness as a viable alternative for speeding

up the aforementioned VLSI CAD operations. We will see that the *Independent-Parallel* and *Master/Worker-Parallel* methods improve the performance of the batch insertion and deletion operations while the *Synchronous-Parallel* method causes additional losses in performance. The parallel algorithms mentioned above can be implemented on any message-passing parallel/distributed system. Our implementation environment is a distributed network composed mainly of SUN workstations.

This thesis is organized as follows. A description of Ousterhout's serial corner stitching algorithms will be presented in Chapter 2. Parallel processing principles and issues will be discussed in Chapter 3. Chapter 4 will present the three sets of parallel algorithms for speeding up the corner stitching insert and delete operations. A performance analysis of the parallel algorithms and experimental results will be presented in Chapter 5. Finally, Chapter 6 makes concluding remarks and discusses future research that can extend from this thesis.

Chapter 2

Rectangular Corner Stitching(RCS)

This chapter discusses Ousterhout's rectangular corner stitching data structuring technique. The first section describes how tiles are represented in the data structure. Section 2.2 summarizes the RCS operations and Section 2.3 outlines the shortcomings of the *point find* operation.

2.1 Representation of Tiles in the RCS Data Structure

Because space that is not occupied by circuit components is explicitly stored, two types of rectangular tiles are represented in the RCS data structure: solid tiles and space tiles. In Figure 2.1, solid tiles are represented by the shaded regions of

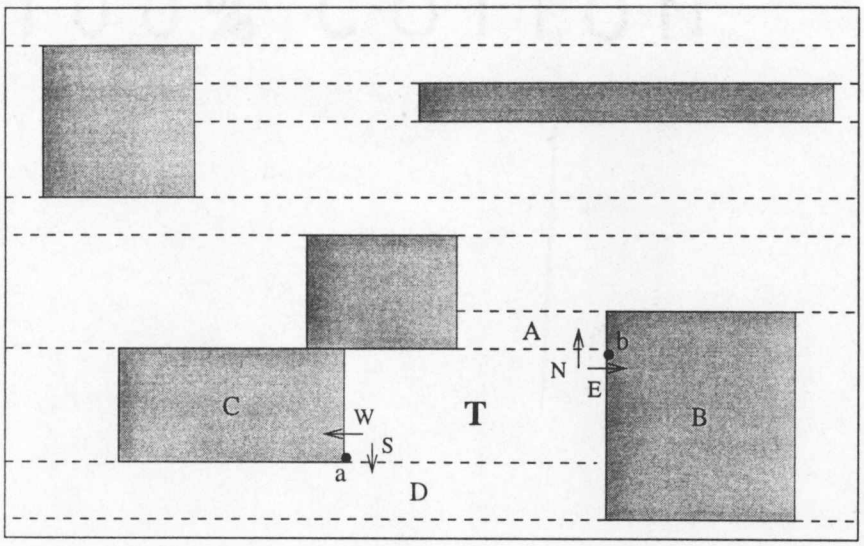

Figure 2.1: Representation of Tiles in a RCS Layout

the layout, while space tiles are represented by non-shaded regions.

Each tile stores a set of four pointers called "corner stitches" (labeled N, E, S, and W in Figure 2.1) that connect the tiles in the layout. As illustrated in Figure 2.1, if T is a tile, then N points to A which is the rightmost tile adjacent to T on its top edge, E points to B which is the topmost tile adjacent to T on its right edge, W points to C which is the bottommost tile adjacent to T on its left edge and S points to D which is the leftmost tile adjacent to T on its bottom edge. These pointers are essential to the execution of the tile manipulation operations and they are the element of the corner stitching data structuring technique that permit operations to have small run-times. Each tile also stores the coordinate of its bottom-left point. Although needed for identification purposes, the top-right point is not stored directly because it can be easily calculated from the bottom-

left points of the tiles pointed to by pointers E and N. In Figure 2.1, tile T's bottom-left point is labeled a and its top-right point is labeled b. To ensure that every point in the tile layout lies within exactly one tile, each tile represented in the layout contains all points on its lower and left edges, but no points on its upper and right edges.

Solid tiles represent the actual VLSI components inserted into the tile layout while space tiles are derived from solid tiles by extending hortizontal lines from each of the four corners of an inserted solid tile until either another solid tile or a boundary of the tile layout is encountered. Figure 2.2 and Figure 2.3 show the partitioning of a layout into solid and space tiles that would occur if Tile A was inserted into the layout. This partitioning method, known as *horizontal maximal striping*, guarantees that no two space tiles share a vertical side. As the data structure is modified by inserting and deleting tiles, the horizontal maximal organization is maintained by splitting and merging techniques described in [7].

2.2 RCS Data Structure Operations

A characteristic that sets RCS apart from other data structures is that it efficiently supports basic as well as more complex operations for tile manipulation in a layout. Most existing layout editing systems that provide complex operations have extremely costly or slow database update operations. The "corner stitched"

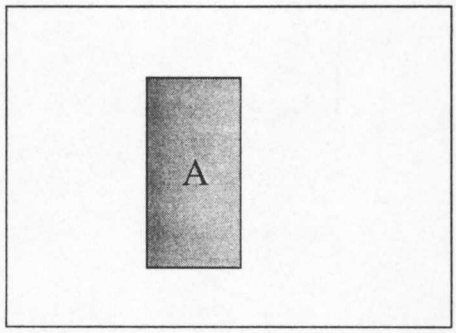

Figure 2.2: Inserted Solid Tile Before Application of Horizontal Maximal Strips

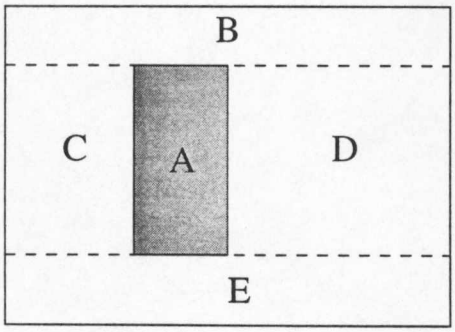

Figure 2.3: Inserted Solid Tile and Space Tiles Derived After Applying Horizontal Maximal Partitioning

representation of tiles in the data structure makes updates to the database a fast and easy task. Each operation of RCS depends solely on information or tiles in the immediate area of the operation. The following operations are provided by the RCS data structure:

1. *Point Find* - Given the coordinate of a point in the tile layout, return the tile containing that point.

2. *Neighbor Find* - Return all the tiles that touch one side [1] of a given tile.

3. *Area Search* - Given a retangular area, return TRUE if no solid tiles intersect that area and FALSE, otherwise.

4. *Directed Area Enumeration* - Given a rectangular area and a starting side, visit and number each tile in an order such that a tile in the given area is visited and numbered only after all the tiles above it and to its left have been visited and numbered.

5. *Tile Insertion* - Given a rectangular area, insert a tile into the layout with the coordinates of the given area.

6. *Tile Deletion* - Delete a tile containing a given point.

7. *Plowing* - When moving a piece of the layout, all solid tiles lying in the path of the motion are moved as well, as if the original piece were a plow.

[1] north, south, east or west tile boundary

8. *Compaction* - Remove all unnecessary space between solid tiles.

9. *Channel Find* - In the corner stitching data structure, channels are, in fact, the space tiles. Therefore channel finding just involves locating the space tiles of interest.

The algorithms for implementing all these operations can be found in [7].

2.3 The Point Find Operation

In this thesis, particular interest is in the point find operation because the insertion and deletion operations make function calls to it. Below is the RCS point find algorithm which, starting with the given tile T, looks for the tile in the layout containing the point (x, y). The choice between lines 6 or 7 and lines 9 or 10 depends on the value of the given (x, y) coordinate.

```
PointFind (T, x, y)
1.  begin
2.          current = T;
3.          while ((x, y) is not contained in current)
4           begin
5.                  while (y does not lie in current's tile y-range)
6.                          current = tile pointed to by current.N; or
7.                          current = tile pointed to by current.S;
8.                  while (x does not lie in current's tile x-range)
9.                          current = tile pointed to by current.E; or
10.                         current = tile pointed to by current.W;
11.         end
12.         return (current);
13. end
```

Two issues that effect execution time are of concern in this algorithm. First, the outer while loop may iterate several times because the horizontal (vertical) motion can cause vertical (horizontal) *misalignment*. Figure 2.4 illustrates the point find algorithm and misalignment. From the start tile, follow the north pointer of all tiles encountered until tile A is reached. Now, follow the current west pointer to tile B. If the west pointer is followed from tile B to tile C, vertical misalignment occurs. As movement continues horizontally through tiles D, E, and finally to F (which is the tile that contains (x, y) in its x-range), our vertical position in the layout is almost equal to the position where we started. As a result of the misalignment, the outer loop of the algorithm must execute again until (x, y) is found. The amount of misalignment depends on the number and relative position of the tiles encountered in the horizontal motion. Greater misalignment results in an increase in the number of iterations of the outer loop.

Secondly, because of misalignment and because the start tile may be very far from the desired point, it is possible that in the worst case, this algorithm may visit every tile in the tile layout. In the average case, the number of tiles traversed is $O(\sqrt{N})$ where N is the total number of tiles in the layout. This is of concern because other data structures can perform this operation in $O(\log N)$ time [4].

Tile insertion and tile deletion are two of the most frequently used operations in layout editing systems. It is also common for designers to perform batches of insertions or deletions into and from the layout. A batch insertion is used to load a file consisting of several thousand tiles into the corner stitching structure at the start of an interactive

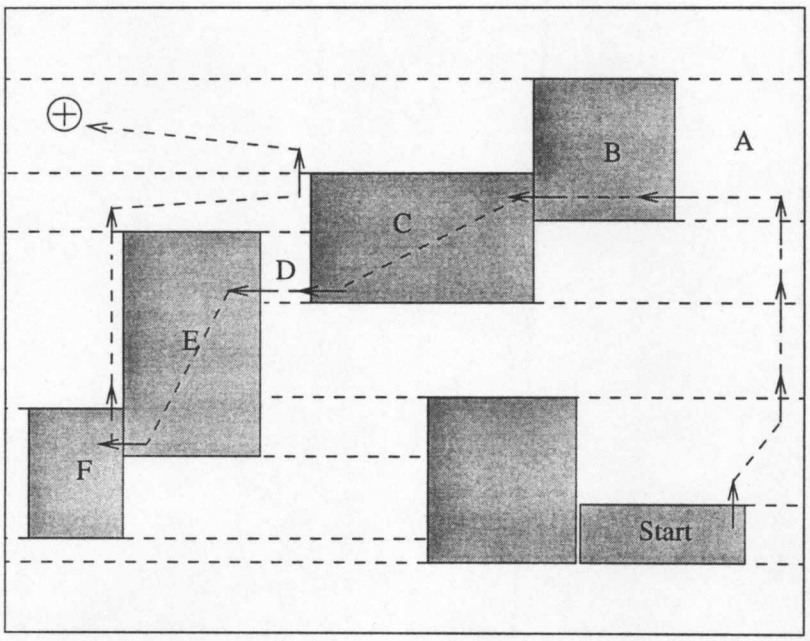

Figure 2.4: Illustration of Point Find Operation and Misalignment

layout editing session while a batch deletion is used to store the contents of the corner stitching data structure into a file at the end of such a session. When performing batch insertion and deletion operations, the time the algorithms spend executing the point find operation accounts for a large part of the total execution time. Therefore, the performance of batch tile insertion and deletion will be significantly improved if the execution time of the point find operation is reduced.

Chapter 3

Parallel Processing

The fundamental principle behind parallel computing is to distribute work among more than one processor. The number of processors can range from as few as 2 to as many as 20,000 as long as they can all operate simultaneously. Parallel computing introduces several new computer architecture criteria that influence the development of parallel applications. For example, processors in a parallel computing environment generally have to interact, so a parallel architecture must provide means for facilitating interprocessor communication. Different programming constructs apply for the different ways an architecture can implement interprocessor communication. The next section discusses these parallel architecture criteria. Parallel computing also introduces new software development issues. Parallel programs must generate multiple streams of instructions and/or data that can be executed in parallel. This is accomplished by applying different programming techniques. Section 3.2 examines key parallel algorithm development

techniques. In this chapter, we will also discuss two major parallel computing environments.

3.1 Parallel Architecture

[2] outlines four basic elements which determine the classification of parallel architectures. These items are instruction execution control, address-space organization, processor interconnection, and processor granularity.

Instruction execution in a parallel machine can either be centralized or distributed. Centralized control, known as *single instruction stream, multiple data stream (SIMD)*, uses a single control unit or processor to spawn work to the other processors. Thus, each processor runs the same set of instructions synchronously. Figure 3.1 illustrates an SIMD architecture. Distributed control, known as *multiple instruction stream, multiple data stream (MIMD)*, implies that each processor has its own control unit and runs its own set of instructions independent of the other processors in the machine. Figure 3.2 is an example of an MIMD architecture. SIMD computers generally need less hardware, require less memory for program execution and are good for programs that synchronize frequently. The major disadvantage of SIMD computers is different processors cannot execute different instructions in the same clock cycle, which can lead to many wasted clock cycles. MIMD machines are usually more complex due to the use of additional control units, but they are more powerful and cost less since they can be constructed using off-the-shelf processors.

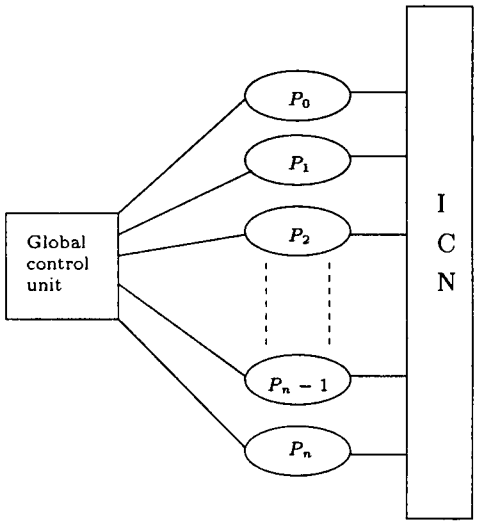

Figure 3.1: Example of a SIMD Architecture. Source: A. Gupta, A. Grama, G. Karypis and V. Kumar. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Co., Redwood City, CA, 1994.

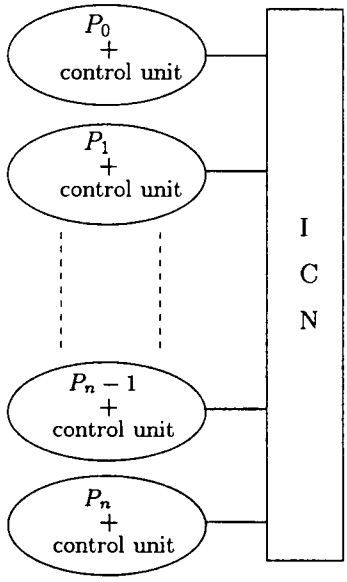

Figure 3.2: Example of a MIMD Architecture. Source: Gupta, Grama, Karypis and Kumar, 1994.

The address-space organization determines how the processors communicate among themselves. The address-space can either be of a *shared* or *message-passing* nature. Each processor of a message-passing architecture has its own local memory and interaction with other processors occurs by passing messages across the interconnection network. Figure 3.3 shows an example of a message-passing architecture. In a shared address-space architecture, processors communicate by reading and writing to blocks of memory that all processors can access via the interconnection network. Figure 3.4 illustrates a shared address-space architecture. Shared address-space architectures are slower because the interconnection network must have a very high bandwidth to accommodate simultaneous memory accesses by all processors. Therefore, message-passing architectures are preferred.

The *interconnection network (ICN)* determines the physical connection among the processors. The ICN can be static which implies communication links do not change or dynamic where communication links between processors can be altered since they are dynamically determined. Typically, message-passing architectures use static ICNs while shared address-space architectures use dynamic ICNs. Some of the important static ICNs are completely-connected, star-connected, ring, mesh, hypercube, tree and graph. The basic dynamic ICNs are bus-based, crossbar switching and multistage. In-depth descriptions of these static and dynamic ICNs are provided in [10, 2].

An architecture is said to be coarse-grained if it is composed of a small number of powerful processors and fine-grained if it is composed of a large number of less powerful

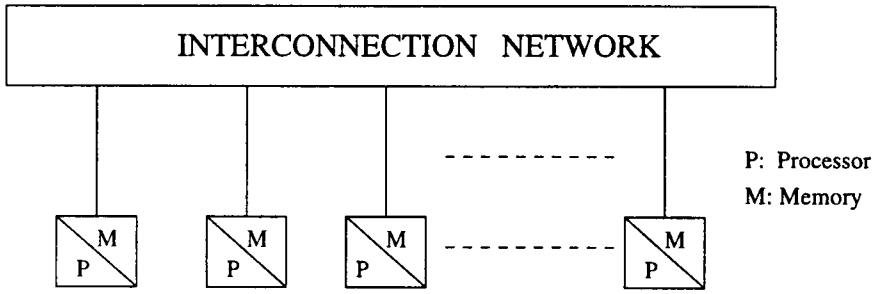

Figure 3.3: Message-passing Architecture. Source: Gupta, Grama, Karypis and Kumar, 1994.

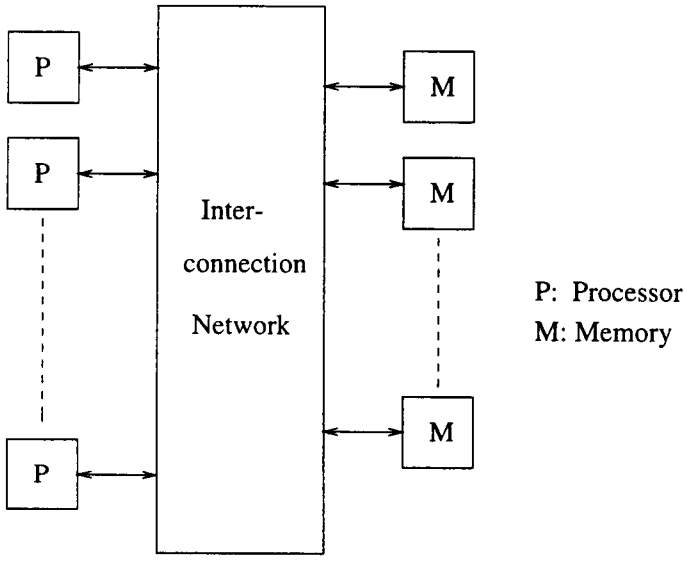

Figure 3.4: Shared Address-space Architecture. Source: Gupta, Grama, Karypis and Kumar, 1994.

processors. There are also medium-grained architectures that lie somewhere between these two extremes usually containing a few thousand processors. Applications with a small amount of concurrency run more efficiently on coarse-grain computers, while applications with a greater amount of concurrency tend to perform better on computers of a finer granularity.

3.2 Parallel Algorithms

Use of the additional computational resources provided by parallel computers, usually requires the development of algorithms that efficiently extract the concurrency in a problem and then map different parts of the problem onto different processors for execution. This mapping of work to processors is a major factor that ultimately determines how well an algorithm will perform on a parallel computer. [3] discusses several methods that can be applied to divide a problem into parts that can be executed in parallel. The decision of which method to use depends on the problem and specifically, the type of parallelism that inherently exists in the problem. The methods are as follows:

1. *Relaxed Algorithm* - each processor works on instructions that can be executed in a self-sufficient manner without synchronization or communication with other processors. Algorithms of this type can result in very large speedups.

2. *Data Partitioning* - The data space is divided into adjacent regions and each region is operated on in parallel by a different processor. Although, interprocessor communication occurs, it is usually minimal because each processor is mainly

concerned with its local data region. Traditionally, this method is very useful in numerical algorithms that deal with large arrays and vectors.

3. *Synchronous Iteration* - Each processor executes the same block of instructions on different parts of the data. But, the processors must synchronize before beginning the next iteration of the instruction block because of data dependencies that exist between the processors.

4. *Replicated Workers* - Tasks that are similar in nature are sent to a pool where a number of worker processors retrieve them and carry out the required task. Execution continues until all the tasks in the pool have been retrieved and serviced by the worker processors.

5. *Pipelined Computation* - Different tasks or phases of a problem are arranged in a structure such as a ring, mesh or hypercube. Each processor of the structure performs its phase of the computation on the data as it passes from one processor to the next through the entire structure.

In methods 2 and 3 above, how the data is partitioned significantly affects the performance of parallel algorithms. [2] describes two types of data partitioning techniques for data parallel algorithms: *Striped partitioning* and *checkerboard partitioning.* These methods are used to partition data that spans a 2-dimensional area. Striped partitioning divides the data area into groups of complete rows or columns and each processor is assigned one of the partitions. Checkerboard partitioning divides the area into smaller

uniform square or rectangular blocks and distributes them among the processors. An illustration of striped partitioning is in Figure 3.5 and an illustration of checkerboard partitioning is in Figure 3.6. The checkerboard method allows for more concurrent streams because it can divide the data among more processors than striping.

3.3 Massively Parallel Processors vs. Networked Parallel Computing

Up to this point, we have referred to parallel computers in a general sense with no distinction concerning the specific types of machines. [1] discusses two popular parallel computing environments that have been instrumental to the growth of parallel computing in general.

Massively Parallel Processors (MPPs) are among the most powerful computers existing today. These machines typically consist of a few hundred to a few thousand processors which are connected to an enormous amount of memory all residing in the same unit. Networked Parallel Computing uses a set of existing general-purpose computers connected by a network as a collective unit to solve single problems. There are software packages that have been developed to allow such a collection of computers to appear as a single machine with any number of the nodes on the network acting as processors in this virtual machine. Although MPPs are more powerful than the networked parallel computing environment, network users that may not have access to a MPP can still solve their problems several times faster than on their standalone computers by us-

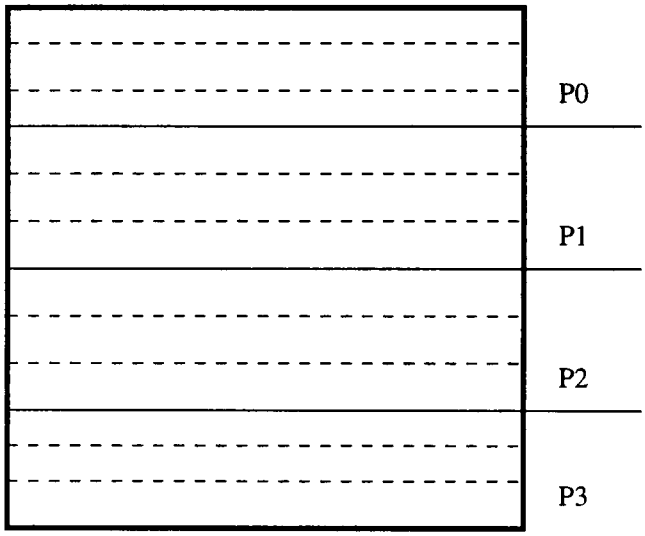

Figure 3.5: Rowwise Striping. Source: Gupta, Grama, Karypis and Kumar, 1994.

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Figure 3.6: Checkerboard Partitioning. Source: Gupta, Grama, Karypis and Kumar, 1994.

ing existing network resources. This is one characteristic that led us to the examination of parallel computing as an option for speeding up batch insertion and deletion corner stitching operations.

Chapter 4

Parallel Implementations

This chapter presents three sets of parallel algorithms that have been developed in an effort to improve the batch insertion and deletion operations. These algorithms are constructed by adding parallel constructs to the serial corner stitching algorithms as implemented in [5]. The parallel constructs distribute the execution of the data structure operations among several processors. All parallel algorithms presented in this chapter assume that the parallel computer on which the algorithms are executed has a *message-passing* architecture. While the intent of this thesis is to develop parallel implementations for the batch insertion and deletion operations of the corner stitching data structure, some of the other operations are also parallelized as needed to accomplish the goal of this thesis.

4.1 Partitioning and Storage of the Corner Stitching Data Structure

Since the tile layout area is a 2-D plane, it is appropriate to use one of the 2-D partitioning methods discussed in Chapter 3 for partitioning the corner stitching data structure among multiple processors. The *horizontal striping* method is used for the parallel algorithms presented in this chapter. Alternative partitioning techniques such as *vertical striping* and *checkerboard* were rejected because they use vertical lines to partition the layout. Figure 4.1 illustrates why the vertical striping method is inadequate. Because *horizontal maximal striping* is used to define space tiles, the insertion of tile A into the layout of Figure 4.1 results in a space tile that spans all of the four partitions. Processors P_0, P_1, P_2 and P_3 are required to maintain their portions (respectively, A_0, A_1, A_2 and A_3) of the space tile. Thus the insertion of a tile into one processor's partition could result in the formation of space tiles in distant partitions. A similar argument shows that the checkerboard method is also unsuitable. As a result of this property, both methods would require considerable additional communication overhead for tile insertion. Tile deletion would also require considerable additional communication overhead. For example, if tile A is deleted from the layout of Figure 4.1, this information would be required at all the other processors resulting in more communicaton overhead. *Horizontal striping*, on the other hand, does not require additional interprocessor communication when generating space tiles because extending horizontal lines from the corners of inserted solid tiles will not cause these lines to cross any partitions. This

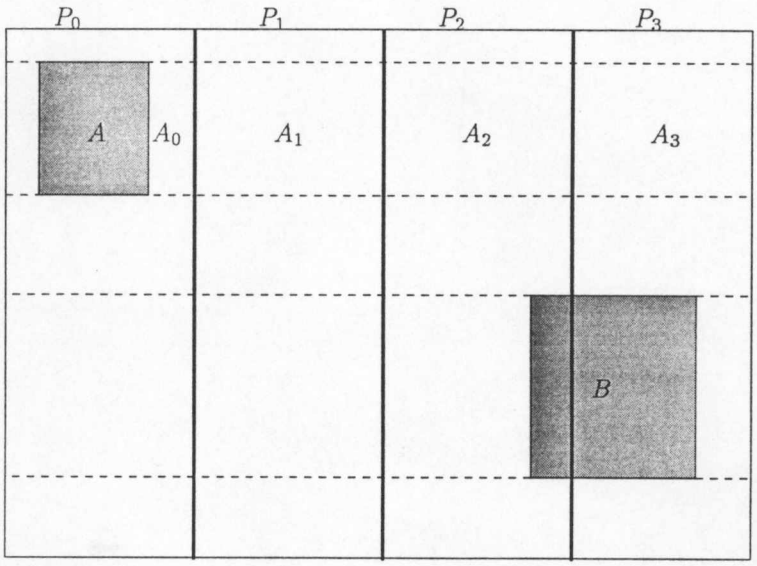

Figure 4.1: Illustration of Vertical Striping Where Space Tiles Cross Partitions When Created

characteristic is illustrated in Figure 4.2. So, the formation of space tiles do not require additional interprocessor communication as with the methods discussed above.

Figure 4.2 also illustrates how the corner stitching data structure is divided among processors. If the number of processors is P and the size of the entire tile layout area is MxN, then each processor is assigned a sub-layout of size MxN/P[1]. As shown in Figure 4.2, processor P_2 is assigned Partition 2. Therefore P_2 is responsible for performing insertions, deletions and other relevant operations to the portions of tiles C, D, and E that lie within Partition 2. Conversely, P_2 is not responsible for any portions of tiles A and B.

Tiles that span multiple corner stitching partitions are split and inserted in parallel

[1]For simplicity, we assume that N is divisible by P

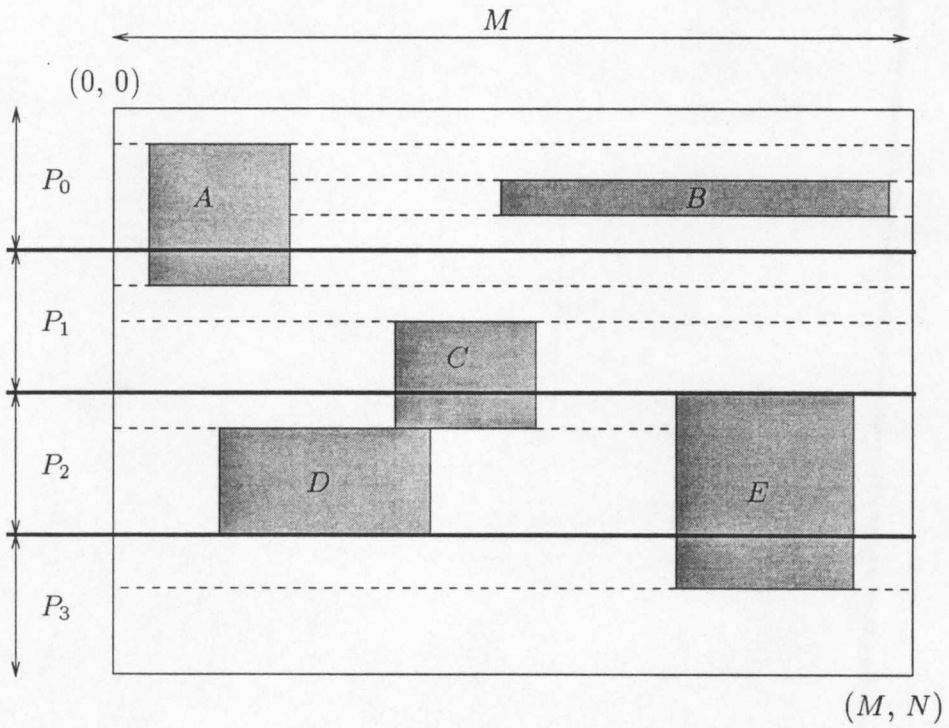

Figure 4.2: Partitioning of Corner Stitching Data Structure Among Four Processors

by the processors that contain a portion of the tile. Figures 4.3 and 4.4 illustrate this process. Tile T is contained in the layout areas for both P_2 and P_3. So, T is split horizontally along the boundary marked *max2/min3* in Figure 4.3 and then each sub-tile produced is inserted by the appropriate processor as in Figure 4.4.

4.1.1 Coordination Data Structure

Splitting tiles that are contained in multiple processors' partitions makes it necessary to incorporate a method of associating the sub-tiles, so that even though they are physically inserted, manipulated and stored as separate entities in separate layouts by different processors, the same behavior occurs that would if they were inserted as a single tile. Therefore, in addition to the four "stitches" and the coordinate of the bottom-left point, each tile represented in the layout has a *tile-ID* field associated with it. Each processor maintains an array, indexed by the tile-ID number, that stores pointers to the tiles in the layout. A pointer is kept for each tile contained within a processor's partition. When inserted into the layout, each tile is assigned a tile ID. If a tile spans multiple partitions, each sub-tile is assigned the same tile-ID by the appropriate processor. So, in Figure 4.4, if the next tile-ID to be assigned is 8, P_2 stores a pointer to T_2 in position 8 of its array, P_3 stores a pointer to T_3 in position 8 of its array while P_0 and P_1 store null pointers in position 8 of their arrays indicating that no portion of tile T is contained in these processor's partitions. So, the array entries for all processors correspond to the same tile.

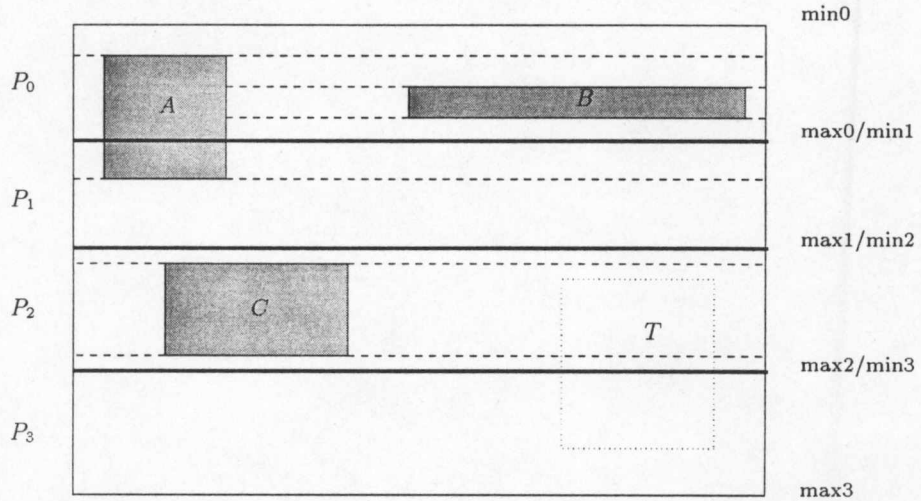

Figure 4.3: The Area of Tile T Overlaps Partitions P_2 and P_3

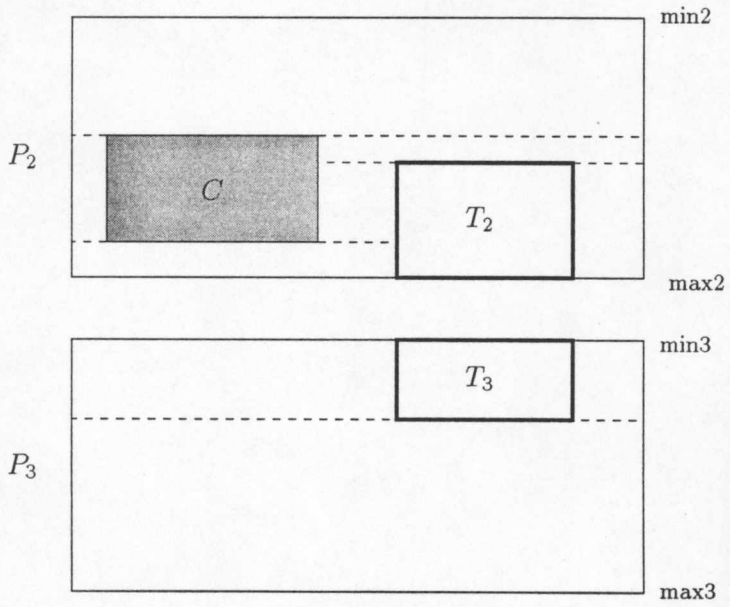

Figure 4.4: Insertion of Tile T in a Parallel Corner Stitching Layout

4.2 Utility Routines

The parallel algorithms for batch tile insertion and batch tile deletion call four basic routines that have been developed to set up a suitable parallel execution environment. This section describes each of these routines. The following variables are used in algorithms in the remainder of this chapter:

y-max-cord-layout - the maximum y-coordinate of the entire tile layout area.

ymin - the minimum y-coordinate of a processor's partition.

ymax - the maximum y-coordinate of a processor's partition.

pid - the identification number assigned to a processor.

start-tile - a randomly or systemically chosen tile already existing in the corner stitching data structure that is used as a starting point for search routines like *point find.*

Other variables used are self-explanatory.

1. *Partition Layout* - Given a processor-id, determine the minimum y-coordinate and the maximum y-coordinate that delimits that processor's partition as labeled in Figure 4.3 as *min n* and *max n* where n is the processor's id number. Processor-ids must be sequentially numbered from 0 to $N-1$ where N is the total number of processors. Each processor must call this routine before attempting to process insertion and deletion batch files.

   Partition_Layout (*pid*)
   begin
   partition-size = *y-max-cord-layout/number-of-proc*;
   ymin = *pid* * *partition-size*;
   ymax = *ymin* + *partition-size*;

end

2. *Modify Tile Area* - If $[p_0, p_1]$ is a tile to be inserted where p_0 represents the lower-left point of the tile area (x_1, y_1) and p_1 represents the upper-right point of the tile area (x_2, y_2), then modify y_1 and y_2 so that the new area represents only the portion of the original area contained in a processor's partition. This routine is used to split tiles that span multiple partitions as in Figure 4.4. After calling this routine, each processor that contains a portion of the tile's area will insert that portion into its corner stitching data structure simultaneously. For clarification purposes, the modified variables will be referred to as my_1 and my_2 in routines that call *Modify_Tile_Area.*

```
Modify_Tile_Area(y1,y2)
    begin
      if (any portion of area [p0, p1] lies in the processor's partition)
         begin
           if (y1 < ymin)
              y1 = ymin;
           if (y2 > ymax)
              y2 = ymax;
         end
      else
         begin
           y1 = not-in-partition constant;
           y2 = not-in-partition constant;
         end
    end
```

3. *Is My Point* - Given the y-value of a point (x, y), determine if the point is in a processor's partition. To ensure every point in the entire tile layout belongs to

exactly one processor's partitition, a processor contains all points on its minimum y boundary, but no points on its maximum y boundary. So, in Figure 4.3, P_0 contains all points on the partition boundary labeled *min0*, P_1 contains all points on the partition boundary labeled *max0/min1*, P_2 contains all points on the partition boundary labeled *max1/min2* and P_3 contains all points on the partition boundary labeled *max2/min3*.

```
Boolean Is_My_Point (y)
    begin
      if ((y ≥ ymin) AND (y < ymax)
         in-partition = TRUE;
      else
         in-partition = FALSE;
      return (in-partition)
    end
```

4. *Parallel Clear* - Before a tile can be inserted into the layout, the area of interest must be clear of solid tiles. If $[p_0, p_1]$ is the area of interest where p_0 represents the lower-left point of the tile area and p_1 represents the upper-right point of the tile area, then this routine determines if the area is clear. *Parallel_Clear* is usually called by each processor after the tile's area has been modified by a call to *Modify_Tile_Area*. The variables my_1 and my_2 represent the modified y-values. If the processor does not contain a portion of the area of interest, it sets its *clear* variable to 1. Each processor that contains a portion of the tile of interest, calls the serial *Area-Clear* routine which determines its value for the *clear* variable. This routine returns 1 if the given area is clear and 0 otherwise. All processors

then send messages to each other to determine whether or not the entire area is clear.

```
Boolean Parallel_Clear (start-tile,x1,my1,x2,my2)
    begin
      if ((my1=not-in-partition-constant) and (my2=not-in-partition-constant))
          clear=1;
      else
          clear=Serial-Area-Clear(start-tile,x1,my1,x2,my2);
      send clear to all other processors;
      for (i=0 to number-of-proc-1)
          begin
              receive clear from another processor and store in recv-buf;
              clear=clear + recv-buf;
          end
      if (clear==number-of-proc)
          return TRUE;
      else
          return FALSE;
    end
```

4.3 General Characteristics of Parallel Algorithms

Before describing the first parallel algorithm, outlined below are general characteristics, some of which are common to all the parallel algorithms presented in this chapter.

Each processor starts the algorithms by first calling the *Partition_Layout* routine. After all processors have executed this routine, the original corner stitching data structure is partitioned into P parts where P is the number of processors. Now there are P individual data structures that are independently maintained by the corresponding processor. These algorithms assume a message-passing architecture, so the information concerning the tiles stored by each processor can be made available to other processors only

via interprocessor communication. Next, the processors begin to process the batch file containing all the tile entries to be inserted or deleted. If the corner stitching operation is *batch insertion*, then each tile entry in the file contains two sets of coordinates: the coordinate of the bottom-left and top-right points of the rectangular area to be inserted. If the corner stitching operation is *batch deletion*, then each tile entry in the file contains the coordinate of any point contained in the layout. The solid tile (if one exists) that contains this point is to be deleted. These tile entries are listed in the batch files one entry per line.

4.4 Synchronous-Parallel Method

This section presents the first parallel approach for the batch insertion and batch deletion corner stitching operations. It is called the *Synchronous-Parallel* method because it is a *Data Partitioning* method that also falls into the *Synchronous Iteration* algorithm category discussed in Chapter 3.

The *Synchronous-Parallel* method requires processors to synchronize for interprocessor communication after each operation in the batch file. If the operation is *batch insertion*, this communication occurs because if a tile, T, overlaps another tile already inserted in the layout then no portion of T should be inserted by any processor. If T spans multiple processors', say P_0 and P_1, then P_0 can not insert the portion of the tile lying within its partition until it ensures that T does not overlap any tiles already inserted in its local corner stitching data structure. In addition, P_0, must ensure that

the portion of T belonging to P_1 does not overlap any tiles already inserted in P_1's data structure. P_1 must also perform the same checks before it inserts its portion of T. This scenario implies that interprocessor communication must occur with the insertion of all tiles in the batch file. If the operation is *batch deletion*, and the tile to be deleted was stored by multiple processors when it was inserted, then the processors will again have to communicate to ensure that all portions of the tile on all the appropriate processors are deleted.

Therefore, at the top level, each processor executes the command sequence below in parallel. This sequence of steps is executed repeatedly until all tile entries in the batch file have been processed. The synchronization step ensures that all processors are inserting (deleting) the same tile at the same time which is necessary because of the dependencies described above.

```
while(not end of batch file)
      begin
1.      read a tile entry from the batch file;
2.      call insertion (deletion) routine;
3.      synchronize;
      end
```

The actual insertions (deletions) occur in step 2 above. In the insertion and deletion algorithms, the processors work in parallel to insert (delete) tiles one at a time as they are read from the batch insertion file. The next subsections describe how this happens.

4.4.1 Batch Insertion

Given the coordinates of the area of interest as read from the current tile entry in the batch file and a start-tile, each processor begins execution of the following algorithm:

```
Synchronous-Insertion(start-tile,x1, y1, x2, y2)
   begin
1.    found-tile = NULL;
2.    not-finished = 0;
3.    if (Area of interest is completely contained in my partition)
4.       begin
5.          if (Serial Area_Clear(start-tile, x1, y1, x2, y2))
6.             found-tile = Serial Tile_Insert (start-tile, x1, y1, x2, y2);
7.       end
8.    else
9.       not-finished = 1;
10.   send not-finished to all other processors;
11.   for (i = 0 to number-of-proc−1)
12.      begin
13.         recv=value of not-finished received by processor;
14.         not-finished = not-finished + recv;
15.      end
16.   if (not-finished == number-of-proc) /*the current tile spans multiple partitions*/
17.      begin
18.         Modify_Tile_Area(y1, y2);
19.         if (Parallel_Clear(start-tile,x1,my1,x2,my2))
20.            begin
21.               if (my1 <> not-in-partition-constant)
22.                  if(my2 <> not-in-partition-constant)
23.                   found-tile = Serial Tile_Insert (start-tile, x1, my1, x2, my2);
24.            end
25.      end
26.   tile-counter++;
27.   if (the processor inserted a portion of the tile stored in found-tile))
28.      begin
29.         found-tile→tile-ID = tile-counter;
30.         tile-array[tile-counter] = found-tile;
31.      end
   end
```

After execution of lines 1-9, one of two scenerios exists: Either the tile was completely contained within one processor's partition and was inserted by that single processor or the tile spanned multiple partitions and needs to be split and inserted by more than one processor. This information is communicated to the processors via the *not-finished* flag, which is sent and received by the processors in lines 10-15.

If *not-finished* equals the *number-of-processors*, then the tile has not been inserted and execution continues with lines 16-25. The *Parallel_Clear* routine is called to ensure the desired area is not already inhabited by other tiles. If this routine returns TRUE, each processor that contains a portion of the tile of interest calls the Serial Tile_Insert routine.

Each processor maintains an array of pointers to inserted tiles. These arrays are updated in lines 26-31. The tile counter corresponds to the last filled entry in the array. All processors increment their tile counters regardless of whether or not they were involved in the insertion of the current tile. If a processor did not insert a portion of the tile, it simply leaves the array entry empty; otherwise it inserts a pointer to the tile in this array entry. This is how tiles that are split are associated among the processors. Each array entry for all processor's data structures correspond to the same tile.

4.4.2 Batch Deletion

Given a point, (x, y) as read from the current tile entry in the batch deletion file, each processor begins execution of the following algorithm:

```
Synchronous-Deletion (x, y)
    begin
1.      if (Is_My_Point(y))
2.          begin
3.              found-tile = Serial_Point_Find (start-tile, x, y);
4.              id = found-tile→tile-ID;
5.              send id to all other processors;
6.          end
7.      else
8.          recv=value of id received by processor;
9.      found-tile = tile-array[id];
10.     if (found-tile)
11.         found-tile = Serial_Delete(found-tile);
12.     tile-array[id] = tile-array[tile-counter];
13.     tile-counter−−;
    end
```

The batch deletion operation of the *Synchronous-Parallel* method also required communication to occur with the deletion of each tile entry in the batch deletion file. Each point in the entire tile layout belongs to exactly one processor. Therefore, exactly one processor will return TRUE from the call to *Is_My_Point* in line 1. This processor finds the tile in its partition containing the point and sends the *tile-ID* of that tile to all the other processors. As a result of lines 1-8, each processor has the *tile-ID* of the tile that needs to be deleted. In the remaining steps of the algorithm, all processors that contain a portion of this particular tile (*tile-array*[*tile-ID*] not empty), delete it from their corner stitching layout.

4.5 Independent-Parallel Method

This approach attempts to use the resources from multiple processors to perform the batch insertion and deletion operations on "more than one" tile entry at a time.

In the previous approach, multiple processors are used to perform operations on the "same" tile entry, but only "one" tile entry at a time. If a processor does not contain a portion of the tile currently being processed, then that processor remains idle until the next tile in the batch file is read or until a tile entry is read that lies within its partition. This idle time is overhead that is eliminated with the *Independent-Parallel* method. This method is in the broad *Data Partitioning* category for parallel algorithms, but it can also be classed as a *Relaxed Algorithm* since it performs the batch insertion and batch deletion operations with close to no synchronization or communication.

In this method, it is assumed that all tile entries in the batch files do not overlap. As a result, more parallelism can be extracted by allowing processors to independently operate on the "portions" of the tiles that are pertinent to them. Each processor independently processes the tile entries in the batch file. If a processor encounters a tile entry that overlaps its partition, it calls the insertion (deletion) routine which performs the appropriate operation on the tile and then it proceeds immediately to reading the next tile entry in the batch file. Again, we assume that in the case of an insertion batch file, no two entries contain areas that overlap each other. As a result, processors can work independently and incur much less idle time than in the previous approach. The top-level of these algorithms for this method is basically the same as the last method

without the synchronization step.

```
while(not end of batch file)
      begin
1.      read a tile entry from the batch file;
2.      call insertion (deletion) routine;
      end
```

The insertion and deletion routine are modified to allow processors to work independently. For the most part, this means elimination of the communication steps in the previous approach. The specific changes are outlined in the next two subsections.

4.5.1 Batch Insertion

Given the coordinates of the area of interest as read from the current tile entry located in the batch insertion file and given a start-tile, each processor begins execution of the following algorithm[2]:

```
Independent-Insertion(start-tile, x1, y1, x2, y2)
   begin
1.    found-tile = NULL or empty;
2.    Modify_Tile_Area(y1, y2);
3.    if ((my1 <> not-in-partition-constant) and (my2 <> not-in-partition-constant))
4.       found-tile = Serial Tile_Insert (start-tile, x1, my1, x2, my2);
5.    tile-counter++;
6.    if (the processor inserted a portion of the tile stored in found-tile))
7.       begin
8.          found-tile→tile-ID = tile-counter;
9.          tile-array[tile-counter] = found-tile;
10.      end
   end
```

[2]This algorithm corresponds to step 2 within the *while* loop of the top-level algorithm above

The difference in this algorithm and the *Synchronous-Parallel* method is that it is no longer necessary to explicitly check to see if a tile entry is completely contained in a single processor's partition. In the previous approach, this was done to prevent the synchronous part of the algorithm from executing when a tile that is completely contained in a single processor's partition is encountered. In this method, the synchronous part of the algorithm has been omitted altogether.

4.5.2 Batch Deletion

Each processor begins execution of the following algorithm:

```
Independent-Deletion ( )
begin
1.  dcount = 0;
2.  while (not end of batch file)
3.      begin
4.          read point (x, y) from the batch file;
5.          execute Pass 1 instruction block;
6.      end
7.  synchronize;
8.  execute Pass 2 instruction block;
end
```

```
PASS 1:
    begin
1.      if (TRUE == Is_My_Point(y))
2.          begin
3.              found-tile = Serial_Point_Find (start-tile, x, y);
4.              del-buffer[dcount] = found-tile→tile-ID;
5.              dcount++;
6.              tile-array[found-tile→tile-ID] = NULL or empty;
7.              found-tile = Serial_Delete(found-tile);
8.          end
9.      tile-counter−−;
    end
```

PASS 2:

```
   begin
1.    send del-buffer array to all other processors;
2.    for (i = 0 to number-of-proc−1)
3.       begin
4.          receive del-buffer array from another processor;
5.          for (each tile-ID in the array buffer received)
6.             if (tile-array[tile-ID] is not empty)
7.                begin
8.                   Serial_Delete(tile-array[tile-ID]);
9.                   tile-array[tile-ID] = empty;
10.               end
11.      end
   end
```

Execution of the batch deletion operation using the *Independent-Parallel* method requires the processors to synchronize or communicate at least once in step 7 of the main portion of the deletion algorithm. In the first pass of the algorithm, each processor will independently examine each entry in the batch deletion file, determine if the point read lies in its partition and if it does, delete the tile containing this point. But, if the tile deleted was split when it was inserted, then the other sub-tile(s) associated with this tile must also be deleted from the corner stitching database that contains each such sub-tile. To handle this scenario, each time a processor deletes a tile, it copies that tile's *tile-ID* field into an array *del-buffer*.

In the second pass of this algorithm, each processor sends and receives a copy of del-buffer to and from all the other processors. For each *tile-ID* in the del-buffers received, the processor determines if it has a non-empty entry in its tile-array structure for that *tile-ID* and if so, it deletes it from its corner stitching data structure.

4.6 Master/Worker-Parallel Method

The algorithm presented in this section provides the third method for performing the batch insertion operation in parallel. The *Master/Worker-Parallel* method can be categorized as both a *Data Partitioning* algorithm where a group of processors perform the same set of insertion instructions on tiles in different portions of the tile layout and as a *Pipelined Computation* algorithm where different phases of the batch insertion operation are performed by different processors. So, unlike the first two methods, this approach extracts some of the functional parallelism that exists in the batch insertion operation of the layout editing problem.

Like the *Synchronous-Parallel* method, this approach handles the case where overlapping tile entries are in the batch file. The *Synchronous-Parallel* method deals with this by requiring all the processors to synchronize and communicate with the insertion of each tile in the batch file. The processors can then collectively determine if the tile of interest overlaps a tile already inserted and discard it if does. Unfortunately, this method yields very high interprocessor communication overhead which is expected to degrade the performance of the batch insertion operation. The *Master/Worker-Parallel* method handles overlap among tiles to be inserted in a batch file in a more efficient manner. The *master* processor maintains an array and a linked-list. The array is used to store nonoverlapping tile entries from the batch file while the linked-list stores tile entries that overlap other tile entries occurring before them in the batch file.

The *Master/Worker-Parallel* approach performs the batch insertion operation in

two phases which are executed in parallel by two different groups of processors. In Phase 1, the *master* processor, P_0, searches the batch file and linked-list for groups of N nonoverlapping tile entries. One at a time, it reads tile entries from the batch file and determines if they are candidates to be included in the nonoverlapping group of tiles. If they are, the tile entries are transferred to the array structure, otherwise they are transferred to the linked-list structure. When the array structure is filled with N nonoverlapping tiles, P_0 sends the array to the second group of processsors. P_0 then repeats this process of finding groups of N nonoverlapping tiles by starting all remaining searches in the linked-list structure. It continues searching for groups of nonoverlapping tile entries until all entries in the batch file and the linked-list have been processed and sent to the second group of processors (*workers*). In Phase 2, after calling the *Partition-Layout* routine, a group of one or more *worker* processors independently begin receiving the arrays of nonoverlapping tiles. These processors then call a slightly modified[3] *Independent-Parallel* method insertion routine to insert the group of tiles into their corner stitching data structures. Each of the processors repeats this step until all groups of tiles have been received and inserted into the data structures.

Below are the algorithms for Phase 1 and Phase 2 of this method:

Phase 1:

```
1.if (processor-id==0)
2.  begin
3.      while (!feof(batch file) or (linked-list <> empty))
4.          begin
5.              batch-count= 0;
6.              top = a pointer to the first entry in the linked-list;
```

[3]The difference is this insertion routine reads tile entries to be inserted from an array rather than a file.

```
7.          if (top <> NULL)
8.            begin
9.              copy top to array structure for nonoverlapping tiles;
10.             top=delete(top);
11.             batch-count++;
12.           end
13.         while ((top <> empty) and (batch-count < N))
14.           begin
15.             overlaps=FALSE;
16.             for (each entry in the array)
17.                   if (top overlaps current array entry)
18.                       overlaps=TRUE;
19.             if (overlaps == FALSE)
20.                   for (each entry prior to top in linked-list)
21.                       if (top overlaps current linked-list entry)
22.                           overlaps=TRUE;
23.             if (overlaps == FALSE)
24.                   begin
25.                       copy top to array;
26.                       top=delete(top);
27.                       batch-count++;
28.                   end
29.             else
30.                   increment top without deleting it from linked-list;
31.           end
32.         while ((!feof(batch file) and (batch-count < N))
33.           begin
34.             read next tile entry from file and check to ensure the data is valid;
35.                   if (data is valid)
36.                       begin
37.                         overlaps=FALSE;
38.                         for (each entry in the array)
39.                               if (file entry overlaps current array entry)
40.                                   overlaps=TRUE;
41.                         if (overlaps == FALSE)
42.                               begin
43.                                   top=pointer to first entry in linked-list;
44.                                   for (each tile entry, top, in linked-list)
45.                                       if (file entry overlaps top)
46.                                           overlaps=TRUE;
47.                               end
48.                         if (overlaps == FALSE)
```

```
49.                                    begin
50.                                        copy file entry to batch array;
51.                                        batch-count++;
52.                                    end
53.                                else
54.                                    insert file entry at end of linked-list;
55.                            end
56.                    end
57.                send array to Phase 2 processors;
58.            end
59. end
```

Phase 2:

```
1.if (processor-id<>0)
2.  begin
3.      while (processor's buffer contains messages)
4.          begin
5.              receive (a message from the buffer);
6.              call (the slightly modified Independent-Parallel insertion routine);
7.          end
8.  end
```

As indicated by line 3 of Phase 1, Processor 0 continues to create groups of nonoverlapping tiles and store them in the array structure until all tile entries in the *batch file* have been examined and until the linked-list structure is empty. The algorithm always begins by checking the linked-list for nonoverlapping entries. Linked-list checking is done in lines 7-31. Note that in lines 10 and 26, the *delete* routine deletes the given node from the linked-list structure and returns a pointer to the node that follows the deleted node in the list structure. Next, if the batch file is not at the end of the file and if the *batch-count* is not equal to N, then the algorithm starts checking the batch file for nonoverlapping tiles. This is done in lines 32-56. Each time N nonoverlapping tiles

are found, Processor 0 executes line 57 which sends the tiles to the Phase 2 processors. At this point, the algorithm loops back to line 3 and repeats this process until the loop conditions are no longer satisfied. The Phase 2 algorithm is self-explanatory.

Chapter 5

Experimental Results

This chapter presents experimental performance comparisons between the serial version of the batch insertion and deletion corner stitching operations and each of the parallel methods for implementing these operations presented in the previous chapter.

5.1 Experimental Environment

The results for the serial version were obtained by executing the serial implementation of the batch corner stitching operations on a SUN SPARCstation 10 for each experimental layout. The results for the parallel versions were obtained by executing each of the parallel implementations of the batch corner stitching operations for each experimental layout on a distributed network of computers containing two to five processors. These processors are random combinations of four SUN SPARCstations 10 and a SUN Sparc 2000. A software package called *PVM, Parallel Virtual Machine* [1] is used to allow

this distributed network of computers to operate as a *message-passing*, *MIMD* parallel computer.

For each parallel approach, the experiment consists of recording the time it takes to perform a batch insertion for each of several randomly generated layouts consisting of 2000 - 60000 rectangular tiles. Next, if applicable to the particular method, the time to delete all the tiles inserted in the first part of the experiment is recorded. The timing information is obtained by using standard UNIX clock commands. In order to closely resemble actual circuits, the tiles are generated so that each tile has a 75% probability of touching a neighboring tile and all tiles are between 20 and 120 units in height and width.

5.2 Experimental Results

The experimental results are summarized in the tables below. The execution times in these tables are expressed in seconds. Table 5.1 contains the insertion and deletion times for a serial implementation of the batch operations. Column 1 indicates the size (number of tiles) of the batch insertion and deletion for which the times are recorded. To assess the amount of improvement or degradation that occurs when using the parallel versions of the batch operations, the results in Table 5.1 are compared to the results of the parallel methods which are summarized in the remaining tables of this chapter. In each of these tables, Column 1 contains the number of tiles in the batch files to be inserted or deleted. Column 2 (respectively, 4, 6, and 8) reports the times for a

Table 5.1: Times for Serial Version of Corner Stitching Batch Operations

No. of Tiles	*Insertion Time*	*Deletion Time*
2000	0.72	0.66
4000	2.09	2.00
6000	3.27	3.23
8000	4.71	4.63
10000	6.13	6.05
15000	9.68	9.43
20000	13.88	13.73
25000	18.90	19.00
30000	24.75	24.97
35000	30.94	31.09
40000	37.30	37.67
45000	45.07	45.67
50000	53.40	54.35
55000	61.52	62.55
60000	71.02	72.90

batch insertion or deletion on a networked parallel computer containing 2 (respectively, 3, 4, and 5) processors. Column 3 (respectively, 5, 7, and 9) is the *speedup* values or the performance gain or loss incurred by executing the appropriate batch operation on a networked parallel computer consisting of 2 (respectively 3, 4, and 5) processors. The formula for speedup is: $S = T_S/T_P$, where T_S is the time required to execute the insertion or deletion operation on a computer with a single processor using the serial implementation and T_P is the time required to execute the insertion or deletion operation on a networked parallel computer using one of the parallel implementations. A speedup of 1 or more indicates performance improvement while a speedup of less than 1 indicates loss in performance.

5.2.1 Synchronous-Parallel Method

Tables 5.2 and 5.3 are the results for the *Synchronous-Parallel* method. As indicated by speedup values of less than 1, this method degrades the performance of the batch operations. The loss occurs because of too much interprocessor communication. For every tile inserted or deleted, interprocessor communication is required. Notice in Tables 5.2 and 5.3 that as the number of processors increase, so does the execution times. Since computation time[1] decreases as the number of processors increase, it can be concluded that the cost of interprocessor communication increases as the number of processors increase.

5.2.2 Independent-Parallel Method

Tables 5.4 and 5.5 are the results for the *Independent-Parallel* method. This method improves the performance of the batch operations by a factor indicated by the speedup values. As the number of processors in the networked parallel computer increase, the time required to execute the batch operations improves. Since very little interprocessor communication is required in this method, if more processors are used to divide the work using the *Data Partitioning* method, then even more time savings can be expected to occur. The point find operation, which is called at least once for each entry in the batch file, is performed faster as a result of the data partitioning. Finding a point now requires a smaller area to be searched. Also, misalignment that can occur while

[1]Computation time represents the amount of time a processor actually spends doing useful work. This time does not include any time due to overhead.

Table 5.2: Insertion Times for Synchronous-Parallel Method

No. of Tiles	*Number of Processors*							
	2		*3*		*4*		*5*	
	Time	*S*	*Time*	*S*	*Time*	*S*	*Time*	*S*
2000	11.06	0.07	12.00	0.06	13.86	0.05	15.54	0.05
4000	22.63	0.09	25.75	0.08	27.82	0.08	31.06	0.07
6000	34.04	0.10	37.78	0.09	39.83	0.08	46.15	0.07
8000	46.98	0.10	50.09	0.09	53.19	0.09	62.02	0.08
10000	58.13	0.11	66.83	0.09	66.49	0.09	81.05	0.08
15000	88.27	0.11	95.54	0.10	99.73	0.10	108.66	0.09
20000	118.90	0.12	133.06	0.10	130.89	0.11	148.79	0.09

Table 5.3: Deletion Times for Synchronous-Parallel Method

No. of Tiles	*Number of Processors*							
	2		*3*		*4*		*5*	
	Time	*S*	*Time*	*S*	*Time*	*S*	*Time*	*S*
2000	10.70	0.06	11.64	0.06	13.64	0.05	14.84	0.04
4000	22.11	0.09	24.03	0.08	26.59	0.08	30.87	0.06
6000	33.84	0.10	37.42	0.09	39.31	0.08	45.07	0.07
8000	44.20	0.10	50.03	0.09	52.79	0.09	63.63	0.07
10000	56.66	0.11	60.69	0.10	62.54	0.10	76.35	0.08
15000	84.37	0.11	96.65	0.10	97.65	0.10	107.15	0.09
20000	114.30	0.12	125.44	0.11	130.52	0.11	145.27	0.09

Table 5.4: Insertion Times for Independent-Parallel Method

No. of Tiles	*Number of Processors*							
	2		*3*		*4*		*5*	
	Time	*S*	*Time*	*S*	*Time*	*S*	*Time*	*S*
2000	0.50	1.44	0.46	1.57	0.44	1.64	0.38	1.89
4000	1.28	1.63	1.02	2.05	0.96	2.18	0.90	2.32
6000	2.02	1.62	1.64	1.99	1.47	2.22	1.34	2.44
8000	2.86	1.65	2.24	2.10	1.99	2.37	1.83	2.57
10000	3.73	1.64	2.82	2.17	2.62	2.34	2.24	2.74
15000	5.93	1.63	4.47	2.17	4.17	2.32	3.56	2.72
20000	8.07	1.72	6.22	2.23	5.45	2.55	4.91	2.83
25000	10.66	1.77	8.13	2.32	7.20	2.63	6.20	3.05
30000	13.90	1.78	10.18	2.43	8.78	2.82	7.72	3.21
35000	17.01	1.82	12.44	2.49	10.60	2.92	9.32	3.32
40000	20.84	1.79	14.80	2.52	12.62	2.96	11.11	3.36
45000	24.29	1.86	18.37	2.45	14.77	3.05	12.80	3.52
50000	28.30	1.89	19.84	2.69	17.12	3.12	14.65	3.65
55000	32.42	1.90	22.83	2.69	19.06	3.23	16.59	3.71
60000	37.01	1.92	26.41	2.69	21.00	3.38	18.39	3.86

Table 5.5: Deletion Times for Independent-Parallel Method

No. of Tiles	*Number of Processors*							
	2		*3*		*4*		*5*	
	Time	*S*	*Time*	*S*	*Time*	*S*	*Time*	*S*
2000	0.44	1.50	0.38	1.74	0.38	1.74	0.38	1.74
4000	1.19	1.68	0.90	2.22	0.81	2.47	0.72	2.78
6000	1.87	1.73	1.40	2.31	1.25	2.58	1.09	2.96
8000	2.64	1.75	1.91	2.42	1.68	2.76	1.50	3.09
10000	3.43	1.76	2.46	2.46	2.18	2.78	1.90	3.18
15000	5.48	1.72	3.93	2.40	3.41	2.77	2.95	3.20
20000	7.56	1.82	5.54	2.48	4.68	2.93	4.04	3.40
25000	10.19	1.86	7.31	2.60	6.27	3.03	5.37	3.54
30000	13.18	1.89	9.33	2.68	7.98	3.13	6.65	3.75
35000	16.32	1.91	11.38	2.73	9.74	3.19	8.14	3.82
40000	19.96	1.89	13.60	2.77	11.18	3.37	9.51	3.96
45000	23.30	1.96	16.44	2.78	13.45	3.40	11.18	4.08
50000	27.38	1.98	18.89	2.88	15.09	3.60	13.43	4.05
55000	31.84	1.96	21.94	2.85	18.37	3.41	14.60	4.28
60000	36.27	2.01	25.73	2.83	19.48	3.74	17.17	4.25

searching for a point is limited to the smaller boundaries of each processors' partition. This can actually result in *superlinear speedup* where the speedup value is greater than the number of processors. Table 5.5 shows that when the number of tiles is 60,000 and the number of processors is 2, the only case of superlinear speedup reported in the experimental results occurs.

5.2.3 Master/Worker-Parallel Method

Tables 5.6 and 5.7 contain results for the *Master/Worker-Parallel* method. Each of these tables correspond to the insertion times required for this method when the *batch size*[2] is 75 and 100 respectively.

The execution time for this method is determined by three factors: time taken by the master processor to form batches (T_m), the time taken by the worker processor(s) to insert tiles into the layout (T_w), and communication time (T_c). An approximate[3] formula for the total execution time based on the three factors might be $T_c + \max(T_m, T_w)$ since T_m and T_w are incurred in parallel. An increase in the number of processors affects these three quantities in the following ways:

1. T_m is unchanged, since the amount of computation performed by the master is unchanged.

2. T_w is expected to decrease based on our findings in Section 5.2.2.

[2]The batch size is the number of nonoverlapping entries the master processor finds before it sends the array containing these tiles to the worker processors to be inserted into the layouts

[3]Approximate because the fact that some communication time may in fact overlap computation time is ignored in the formula

Table 5.6: Insertion Times for Master/Worker-Parallel Method with Batch Size of 75

No. of Tiles	*Number of Processors*							
	2		*3*		*4*		*5*	
	Time	*S*	*Time*	*S*	*Time*	*S*	*Time*	*S*
2000	1.08	0.67	0.73	0.99	0.79	0.91	0.82	0.87
4000	2.65	0.79	1.78	1.17	1.54	1.36	1.59	1.31
6000	4.69	0.70	2.79	1.17	2.44	1.34	2.52	1.30
8000	6.20	0.76	3.78	1.25	3.15	1.50	3.58	1.32
10000	8.05	0.76	4.91	1.25	4.27	1.44	4.44	1.38
15000	12.98	0.75	7.62	1.27	6.62	1.46	6.99	1.38
20000	18.02	0.77	10.80	1.29	9.20	1.51	9.95	1.39
25000	23.96	0.79	14.31	1.32	10.01	1.89	10.71	1.76
30000	29.48	0.84	18.38	1.35	13.10	1.89	14.11	1.75
35000	35.81	0.86	22.88	1.35	15.18	2.04	15.50	2.00
40000	42.66	0.87	26.61	1.40	16.98	2.20	17.30	2.16
45000	48.16	0.94	29.73	1.52	20.13	2.24	20.38	2.21
50000	55.80	0.96	34.78	1.54	21.75	2.46	21.88	2.44
55000	63.32	0.97	38.85	1.58	25.39	2.42	25.50	2.41
60000	72.80	0.98	42.49	1.67	28.21	2.52	29.19	2.43

Table 5.7: Insertion Times for Master/Worker-Parallel Method with Batch Size of 100

No. of Tiles	*Number of Processors*							
	2		*3*		*4*		*5*	
	Time	*S*	*Time*	*S*	*Time*	*S*	*Time*	*S*
2000	1.25	0.58	0.78	0.92	0.74	0.97	0.81	0.89
4000	2.72	0.77	1.66	1.26	1.60	1.31	1.60	1.31
6000	4.34	0.75	2.60	1.26	2.47	1.32	2.44	1.34
8000	5.68	0.83	3.80	1.24	3.08	1.53	3.32	1.42
10000	7.82	0.78	4.59	1.34	4.08	1.50	4.42	1.39
15000	12.71	0.76	7.38	1.31	6.17	1.57	6.56	1.48
20000	16.99	0.82	10.86	1.28	7.76	1.79	9.08	1.53
25000	24.42	0.77	14.71	1.28	10.39	1.82	10.41	1.82
30000	30.39	0.81	17.04	1.45	13.28	1.86	13.21	1.87
35000	33.47	0.92	22.56	1.37	14.47	2.14	16.88	1.83
40000	41.51	0.90	26.00	1.43	17.52	2.13	20.68	1.80
45000	47.60	0.95	27.40	1.64	21.53	2.09	25.79	1.75
50000	53.59	1.00	33.96	1.57	22.17	2.41	31.08	1.72
55000	62.49	0.98	39.04	1.58	25.41	2.42	23.87	2.58
60000	72.09	0.99	43.89	1.62	29.81	2.38	25.60	2.77

3. T_c is expected to increase based on our findings in Section 5.2.1.

When two processors were used to perform the batch insertion, the execution times were slightly higher than those of the serial implementation. This is to be expected since the solitary worker processor is responsible for the entire layout and performs all the insertions. Since the layout is not partitioned in this case, none of the benefits of reduced search area and reduced misalignment during the point find operation are to be expected.

The improvement in execution times when three processors are used instead of two and when four processors are used instead of three may be attributed to an improvement in T_w which offsets any degradation in T_c. The degradation in execution times when five processors are used instead of four may be attributed to a worsening of T_c coupled with diminishing improvements in T_w. The diminishing improvements in T_w can be observed in most entries of Table 5.4. For example, three processors insert 60,000 tiles 10.6 seconds faster than two processors, four processors perform the same computation 5.41 seconds faster than three processors and the improvement by using five processors instead of four processors in only 2.61 seconds.

Figure 5.1 is a graphical representation of these results for 25,000 insertions and a batch size of 75.

The optimal performance achievable by the *Master/Worker-Parallel* method is not only dependent on the number of processors used to perform the insertions, but also on the batch size. As the batch size N increases, T_m is expected to increase linearly with

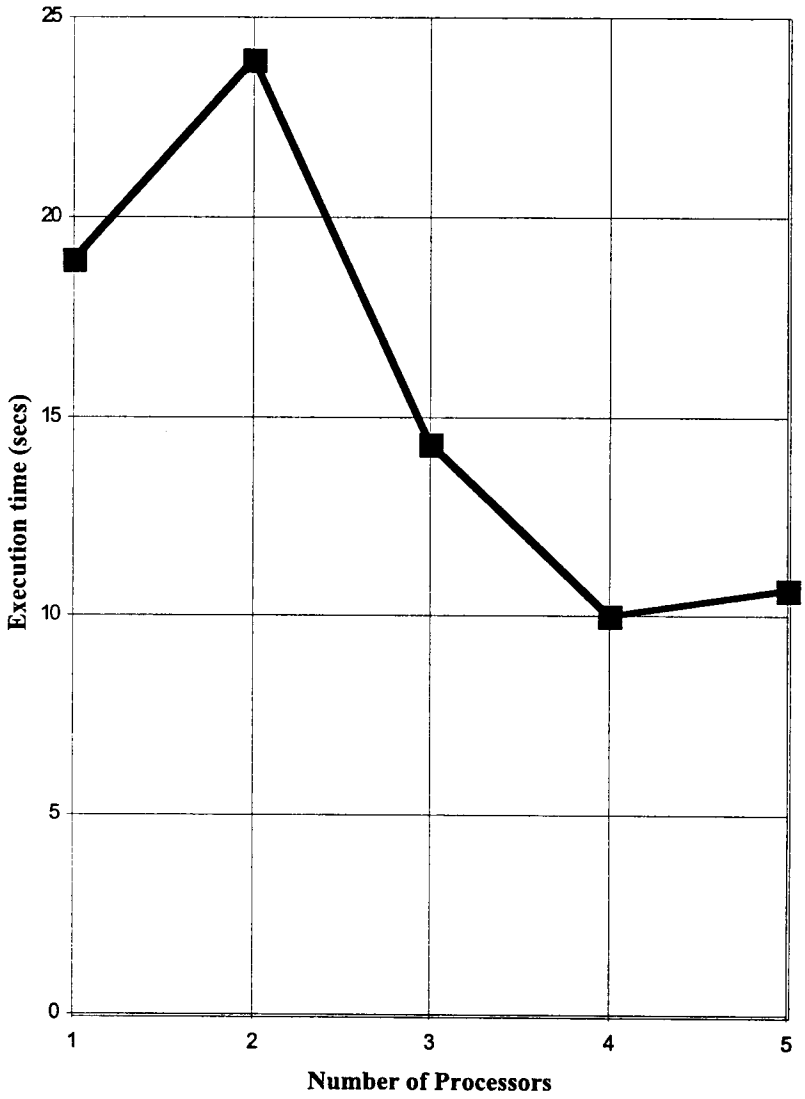

Figure 5.1: Execution Time per Number of Processors

N: the amount of time required to create a batch of size N is $O(N^2)$ since all pairs of tiles in the batch are checked to make sure that there are no overlaps. The number of batches is approximately $O(M/N)$, where M is the total number of tiles to be inserted. Hence, the total time for batch creation is $O(M/N * N^2) = O(MN)$. This analysis does not take into account the additional time spent on tiles that overlap a tile in the batch array. It also ignores the fact that some of the later batches may have less than N tiles.

An increase in batch size reduces the number of data broadcasts from the master to the workers, but does not effect the total amount of data transmitted during the course of the algorithm. Since the communication start-up time is usually quite large, it is expected that a reduction in the total number of data broadcasts will result in a reduction in T_c. An increase in batch size should not effect T_w.

Our experiments show that an increase in batch size initially lowers the execution time but later causes the execution time to increase. The initial improvement is attributed to the decrease in T_c dominating the increase in T_m. The later degradation in performance is attributed to the increase in T_m dominating the decrease in T_c. The optimal batch size varies for different number of processors. Figure 5.2 shows that the optimal batch size for 3, 4 and 5 processors is approximately 60, 75 and 100, respectively.

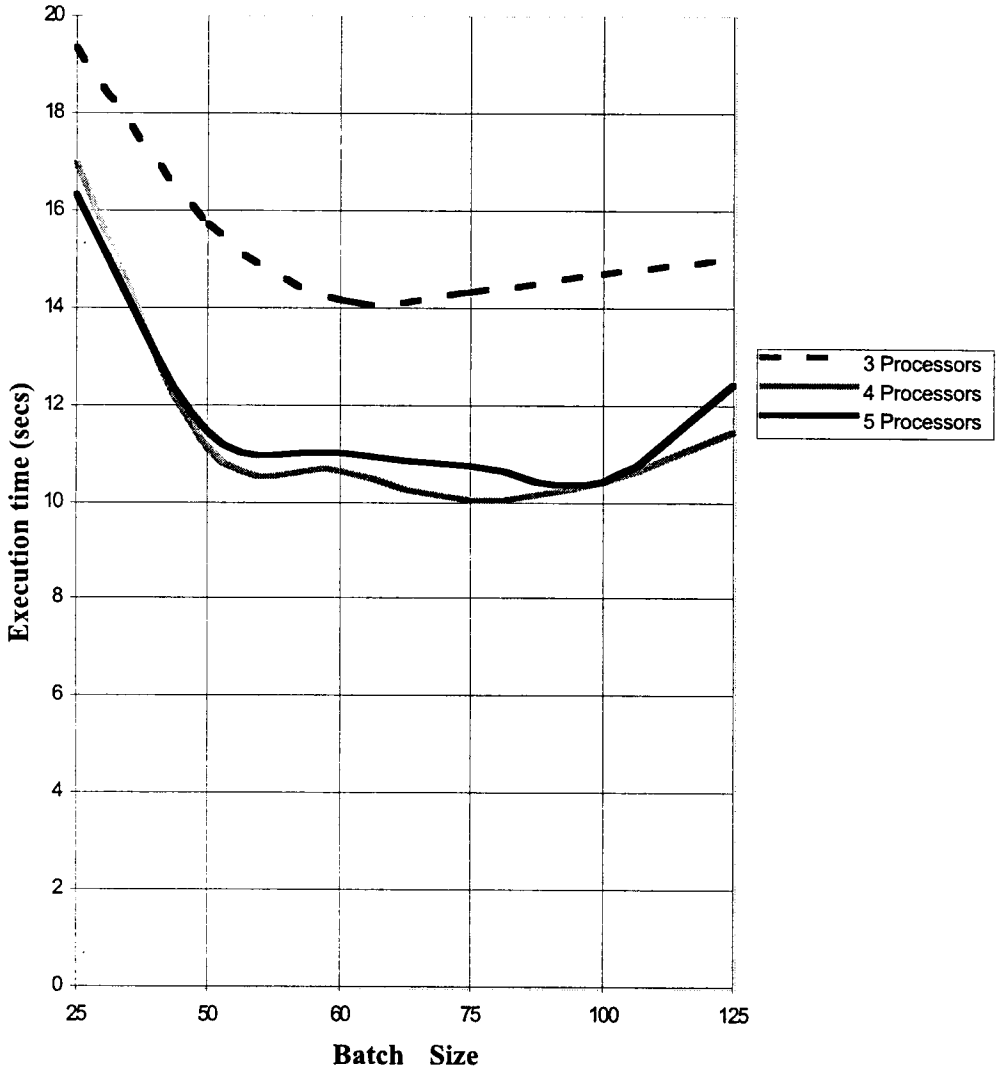

Figure 5.2: Execution Time per Batch Size

Chapter 6

Conclusions

The experimental results presented in the previous chapter suggest that parallel algorithms that do not require an unreasonable amount of communication can significantly improve the performance of the batch insertion and deletion corner stitching operations. Therefore, it can be concluded that the *Independent-Parallel* and *Master/Worker-Parallel* methods are viable parallel solutions for improving the performance of the batch corner stitching operations while the *Synchronous-Parallel* method is not.

Future areas of research that can be pursued from this thesis are the development of parallel algorithms for other batch corner stitching operations and use of the presented parallel algorithms in L-shaped corner stitching, Trapezoidal corner stitching and other non-rectangular implementations of the data structure.

Bibliography

Bibliography

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. *PVM Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[2] A. Gupta, A. Grama, G. Karypis, and V. Kumar. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Co., Redwood City, CA, 1994.

[3] Bruce Lester. *The Art of Parallel Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

[4] David Marple, Michiel Smulders, and Henk Hegen. "Tailor: A Layout System Based on Trapezoidal Corner Stitching". *IEEE Transactions on Computer-aided Design*, 9(1):66–90, 1990.

[5] George J. Blust. *Corner Stitching for L-shaped Tiles.* Master's Thesis in Computer Science. University of Tennessee, Knoxville, TN, 1994.

[6] J. Rosenberg. "Geographical data structures compared: A study of data structures supporting region queries". In *Proc. 1985 IEEE Int. Conference Computer-aided Design*, pages 53–67, 1985.

[7] John K. Ousterhout. "Corner Stitching: A data structuring technique for VLSI layout tools". *IEEE Transactions on Computer-aided Design*, 3(1):87–100, 1984.

[8] R. L. Brown. "Multiple storage quad trees: A simpler faster alternative to bisector list quad trees". *IEEE Transactions on Computer-aided Design*, 5:413–419, 1986.

[9] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Hilger Publishing Co., Bristol, England, 1981.

[10] Terry Fountain. *Parallel Computing: Principle and Practice.* Cambridge University Press, New York, NY, 1994.

Vita

Erica Dionne Wilson was born in Jackson, Mississippi on November 12, 1971 to Mr. and Mrs. Herman Wilson. She attended Provine High School in Jackson, Mississippi where she graduated as valedictorian of her class in June, 1989. In August of the same year, she enrolled in Jackson State University. In May, 1994, the Bachelor of Science in Computer Science was conferred to her by Jackson State University.

After receiving a fellowship from the GEM Consortium, Erica elected to enroll in the Master's program in Computer Science at The University of Tennessee Space Institute in August, 1994. She received the Master's degree in May, 1996.