Mask Verification on the Connection Machine

Erik C. Carlson and Rob A. Rutenbar

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Parallel mask verification algorithms have been developed for the Connection Machine, a massively parallel processor with up to 64K processors. We discuss the design and implementation of algorithms for several essential primitives: generation of completely-intersected mask data, mask-to-mask Boolean operations, labeling of connected regions, and identification of width and spacing violations. Performance results from experiments on a 16K-processor machine are presented. Speedups between 40 and 240 over a VAX 11/785 have been measured.

1. Introduction

Mask verification is often cited as a potential candidate for acceleration on parallel hardware. In part this is because individual mask checking operations can be computationally expensive, but also because a complete verification task involves many individual checking operations on different mask layers. Alternatives to traditional flat checking, such as hierarchical checking, interactive checking, and correct-by-construction methodologies have all been implemented with some degree of success. Nevertheless, demand for complete, flat mask checking, in particular mask extraction, remains surprisingly strong. One potential reason is that some alternative checking methodologies impose burdensome restrictions on mask artwork, for example, Manhattan-only geometry, or cell abutment constraints. With shrinking device sizes, increasingly complex, yield-optimizing design rules, and the increasing presence on chip of both digital and (previously separate) analog circuitry, demand for fast, accurate, general-purpose mask verification and extraction will only increase.

Unfortunately, parallelism in mask checking tasks has not been aggressively exploited in practice. One problem is that early approaches to parallel checking relied on unrealistic, over-simplified representations for masks such as bit maps [1,2,3]. Previous special-hardware approaches have also also been limited in that they typically accelerate only a portion of the solution process. Mask verification is not a single, homogeneous task, but is instead a complex sequence of different subtasks. This makes it extremely unlikely that a cost-effective mask checking system could be implemented *completely* in dedicated hardware.

More recent approaches have focussed on exploiting course-grained parallelism on general-purpose multiprocessors. For example, in a complex suite of verification operations, independent checking subtasks that can proceed in parallel can each be assigned to a separate processor [4]. This approach works well, but is limited by the number of separate, non-interacting subtasks that can be found in any particular verification operation. Another option is to divide the layout to be checked into large pieces, run conventional checking tools on each piece on a separate processor, and then attempt to glue the results back together [5,6,7]. A general concern here is that with too many small pieces of layout, reconstructing the final answer from the individually verified pieces may be dominated by the time to reconcile complications at the boundaries (e.g., individual devices split across separate processors). Although workable and practical, such coarse-grain schemes give little insight into the *limits* of extractable parallelism in large-scale mask verification problems.

Our central concern in this paper is to address the feasibility of mask verification on *massively parallel* multiprocessor architectures. Such machines, notably the Connection Machine System [8], allow thousands of individual processors to cooperate on specific computational tasks. Recent studies of other CAD algorithms on Connection Machines have shown promise [9,10,11]. However, such machines require substantially new strategies for mapping mask verification tasks onto available computing resources. For example, simple coarse-grain schemes which bind a few tens of independent subtasks to individual processors are unsuitable here: using only 10 out of 10,000 processors is unacceptably wasteful. For the mask checking problem, algorithm partitioning schemes of which we are aware rely almost negligibly on the *cooperation* of multiple processors to perform single, primitive checking tasks. This style of parallelism is essential in order to fully exploit massively parallel machines.

This paper reports preliminary results from several mask checking algorithms running on a 16K-processor CM-2 Connection Machine. The starting point for these studies was our previous work on special-hardware for scanline algorithms. In [12] we argued in favor of hardware that operated directly on flattened edge data (a realistic representation for mask checking problems), and that supported several checking tasks related within a common algorithmic framework. Although this hardware was never built, many of the underlying ideas concerning fine-grain parallelism in edge-based algorithms have proven to be useful in the context of a fine-grain parallel processor such as the Connection Machine. In addition, portions of the verification task that were essentially unimplementable in our special hardware, notably the "glue" tasks that tie major processing steps together (e.g., sorting operations) can be handled gracefully and efficiently on a Connection Machine.

The remainder of the paper is organized as follows. Section 2 reviews the architecture and programming model for the Connection Machine. Section 3 describes a basic algorithmic framework for edge-based mask checking algorithms on a massively parallel machine, and then describes parallel algorithms for Boolean mask combinations, region numbering for connectivity analysis, and width/space checking. Section 4 briefly describes some Connection Machine implementation issues. Section 5 presents measure-

ments for these algorithms running on a 16K-processor machine, and compares them with the performance of representative serial algorithms. Finally, Section 6 presents some concluding remarks.

2. Connection Machine Overview

Parallel processor systems can be broadly characterized by examining their two essential components: the individual processors, and the interprocessor communication medium. Multiprocessor systems with roughly 10 to 100 processors are now commercially available. These machines mostly employ processors based on mature microprocessor technology, interconnected by busses, multi-stage interconnection networks, or message-passing links.

In contrast to these smaller machines, the Connection Machine employs a substantially different architecture [8]. A major difference is the size of the machine: up to 64K small, bit-serial processors, each with local memory. The current version of the machine, the CM-2, has 64 Kbits of memory per processor, and also includes a dedicated floating point unit for every 32 processors. Another fundamental difference is that the Connection Machine executes instructions in a SIMD fashion, each processor executing the same instruction from a single thread of control. Smaller multiprocessors are universally MIMD machines, each processor following its own arbitrary thread of control.

Connection Machine processors communicate over a routing network whose details can be regarded as hidden from the programmer. Each processor can execute a **Send** or **Get** instruction to transfer data to or from any other processor. There are no limitations on the number of processors that can access this network simultaneously, or on the nature of the data moved across the network.

Three characteristics of the Connection Machine impact the manner in which it is programmed: the large number of processors, the relative inexpensiveness of communication, and the single thread of control. In smaller machines, the emphasis is on *decomposing* the critical features of a problem onto a limited number of physical processors. On the Connection Machine, an alternative strategy, referred to as *data parallelism* [13], is more natural. Processors are sufficiently numerous that each of the individual data objects that comprises a problem can be given its own processor; operations can thus proceed in parallel on each data object. When the number of data objects exceeds the number of physical processors, *virtual processors* can be employed. The physical memory on each node is partitioned into different segments, each of which holds the data for a virtual processor *sharing* that physical processor. Virtual-to-physical swapping is handled automatically.

These basic capabilities can also be combined in complex ways in the construction of programs for the machine. Typical programming style involves *selecting*, as a result of some detailed computations, a subset of processors to become active; only these selected processors then execute instructions from the SIMD instruction stream. The selected processors then typically pass results of their computations to some other processors (real or virtual), with the routing network deducing precisely where to transport the data. The process of selecting, computing, and transporting can then be repeated.

An important class of operators is available to programmers for dealing with linear lists of data objects, each of which may be stored on a different processor [13]. These are extremely useful to us because our mask checking algorithms rely heavily on processing many linear edge lists in parallel. **Scan** operators are applicable when successive elements in the list are stored in consecutively numbered processors (each processor, real or virtual, has a unique ID number by which is may be addressed). For example, given a list of data items D_i, indexed from *0* to *N-1*, a **plus-scan** operation deposits in item i of the list the sum $\sum_{i=0}^{k} D_i$. A related operation, **copy-scan,** can then be used to copy a particular sum value, say the sum in element *j*, back down the list to elements *0, ... , j-1*. If the D_i represent pointers in a linked list (where a pointer is just a processor ID number), a similar set of operations can be used to deduce the end of the list, i.e., to place in each data item a pointer to this last item. Unlike the **scan** operators, these more general list manipulation operations do not require storage of data in consecutive processors. All these operations can be performed in $O(\log L)$ time, where L is the length of the list. Note that we can also process several different lists in parallel; these individual lists, called *segments*, are first concatenated into one large linear list, and then all segments are processed in parallel.

Because of the general-purpose routing network, sorting operations can also be performed with considerable efficiency. Sorting here has time complexity $O(\log^2 N)$ for N elements [13]. For example, using a radix sort, a CM-2 can sort 64K 32-bit quantities in roughly 30 ms. As will be described in the following sections, these parallel list processing and sorting abilities play a key role in parallel mask checking algorithms.

3. Algorithmic Development

This section first describes the basic framework for our parallel mask checking algorithms. We then describe new parallel algorithms to form completely intersected mask data, and to perform mask-to-mask Boolean operations, single-mask region labeling, and space and width checking.

3.1. Scanline Parallelism: Basic Considerations

Several assumptions underlie this work on parallel algorithms. First, the emphasis has been on the basic design of the primitive checking operations from which complete verification systems can be constructed. Hence, we examine primitives such as Boolean mask combinations, region numbering, and distance checking. Second, we employ a flat, edge-based representation, and dedicate a separate Connection Machine processor to each edge; even modestly large machines have sufficient processor and memory resources to make this practical. Third, we rely on a scanline *style* for our algorithms. Scanline algorithms exist for many mask checking primitives, have excellent asymptotic complexity (for serial processors) and have been widely used in industrial applications [14,15]. In particular, scanline algorithms can accommodate non-Manhattan geometry. However, critical serial bottlenecks in conventional scanline algorithms prevent any trivial mapping onto a massively parallel machine. We retain some broad aspects of the *style* of scanline algorithms, but we restructure them to exploit the large amount of parallelism available in the machine. Finally, we restrict ourselves initially to only Manhattan geometry to simplify some aspects of our implementations. Nevertheless, the basic scanline-style framework adopted will support oblique geometry; indeed, this was a primary motivation in its development.

A critical feature of flattened edge data is that verification computations are highly localized; individual edges only interact with their close neighbors. Algorithms that process mask geometry can be characterized by their mechanisms for exploiting this locality. Scanline algorithms are so-named because they sweep a virtual line, called the *scanline*, across the individual edges comprising the mask; only those edges

that encounter the scanline need to be considered. By maintaining appropriate state information between successive scanline locations, most of the actual checking operations simply involve stepping through the active edges on the scanline in the appropriate order. The historical motivation for scanline algorithms is that with limited memory resources, it is essential to minimize the amount of edge data resident in memory at any time; scanline algorithms require only one scanline's edges to be present in memory.

However, for our purposes, this style of checking has a serious disadvantage: one scanline must be completely checked before its successor can be started. This is because both the location of the next scanline stop, and the set of edges intercepted by this next stop, are determined by computations done in the previous scanline. Because of this sequential bottleneck, a naive mapping of a scanline algorithm onto a Connection Machine yields minimal parallelism. To bypass these sorts of problems, we introduce a new parallel preprocessing phase for the edge data. The idea is to find *all* scanline stops in parallel at the start of checking, and then to process each of the resulting scanlines in parallel. The following sections describe the preprocessing algorithm, and the subsequent checking primitives that rely on it.

3.2. Preprocessing: Complete Edge Intersection

A set of edges is said to be in *completely intersected* form if all edge intersections occur only at edge end points. This form is attractive because it eliminates the need to check for edge intersections as possible locations where the scanline must stop to examine edges. For our purposes, we further require that all edges that cross a scanline stop be split at their point of intersection. If this is done, then processing all scanlines in parallel is straightforward: for vertical scanlines and Manhattan geometry, each unique X coordinate of an edge end point *is* a scanline stop. Thus, in general if we can identify *all* edge end points and edge-to-edge intersections, the set of unique X coordinates found is precisely the set of scanline stops. By then splitting each edge that crosses any of these coordinates into two separate edges, we can bring the entire mask into completely intersected form.

Assume there are N total edges. First, the edge list is read into the Connection Machine and each edge is mapped onto a unique (possibly virtual) processor. Like some classical scanline algorithms [16,17], we only explicitly represent the non-vertical edges, since the vertical edges can be reconstructed if we store whether the regions above and below each horizontal edge represent transparent or opaque parts of the relevant mask. Long edges are split by repeatedly bisecting them until they are sufficiently short; one of the two new edges created is allocated to a new processor. The reason for this step will be discussed shortly.

Next, we sort all edges on their minimum X coordinate. The sorting algorithm deposits the sorted edges in a single linear list mapped onto consecutively numbered processors. The next task is to extract from this list just the unique X coordinates. In parallel, each processor examines its own X, and the X of its higher-numbered neighbor. Those processors which see a different value in their neighbor hold the unique representative X for all edges with X as an end point. At this point, we can regard all edges with a common minimum X coordinate as constituting a scanline.

The next step is to split all the edges that *cross* one of these unique X scanline stops. If we number the vertical scanlines left to right, this splitting can be done by passing the X coordinate in scanline i to all edges in scanline i-1, performed in parallel for all scanlines. Each edge in each scanline can then in parallel determine if it crosses it's neighboring scanline, and those edges which do are split at that X value, and the newly created edge is sent off to a new processor. If we repeat this process of passing X coordinates, scanline i-2 now gets the X location of scanline i (again, this happens across all scanlines), and its crossing edges can be split. This process repeats until no scanline i holds an edge that can possibly intersect the X location of scanline $i+k$ being considered. The reason for earlier splitting long edges is now apparent: it reduces the number of scanline intersections that any given edge is likely to encounter, and accelerates this process.

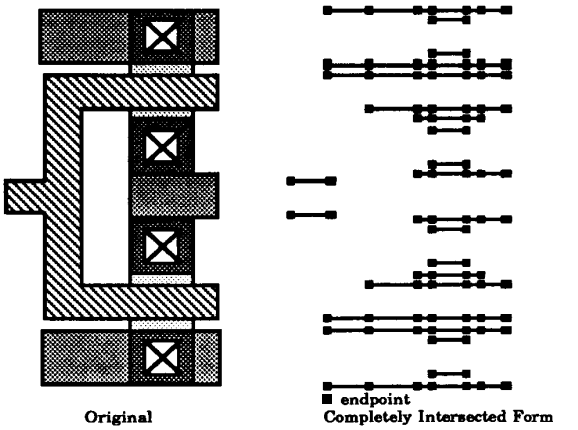

Figure 1. Complete Intersection of Mask Layers

This final, larger set of edges is *almost* in completely intersected form, however, we have only checked potential intersections with the minimum X coordinate of each edge: we must now check for intersections with the maximum X coordinate. This process is directly analogous to that for splitting based on minimum X coordinates, except that the preliminary sorting operation now includes the newly split edges from the previous phase, and scanline coordinates propagate in the opposite direction. We require that the final edge data exist in the form of a single linear list, sorted in two-dimensions, i.e., sorted on X, and among edges with identical X, sorted on Y. Hence, a sorting operation is performed, and the resulting list items are deposited on consecutively numbered processors.

The complexity of this task is rather difficult to analyze because the volume of edges changes after each of the splitting phases. Informally however, we can analyze it as follows. The total time is always dominated by the IO, which requires $O(N)$ time for N edges. The propagation of X coordinate values from scanline i to scanline i-k can be done with scan operators in time $O(\log E)$, where E is the maximum number of edges in any scanline, and E is itself $O(N^{1/2})$ expected edges [17]. Splitting any edge is $O(1)$, but finding new processors for all these simultaneously created edges takes $O(\log S)$ time for a total of S split edges. For N total edges in the scanlines, there are at worst $O(N^{1/2})$ iterations of this step, since at most, each scanline must examine the coordinates of all other scanlines, and there are $O(N^{1/2})$ expected scanlines [17]. In practice however, the number of iterations is small since long edges are split in the first step of the algorithm. Each sorting operation requires $O(\log^2 N)$ time, where N is the number of resulting intersected edges. With N input edges, in pathological cases the completely intersected form can be as large as N^2 edges, i.e., each edge intersects *all* other edges. However, in practice, this size tends to be considerably smaller, much closer to N than to N^2. An example of the generation of the complete intersected form of mask layout is shown in Fig. 1.

This edge format is the assumed starting point for the algorithms described in the following sections.

3.3. Boolean Operations

Boolean operations can compute unions, intersections, complements, etc., for mask layers. Because we start from a totally intersected representation, the edges that the bound regions in the output of a Boolean operation are also present in the input: we never have to split an edge to form an output boundary. We rely here on a serial scanline algorithm due to Lauther [16]. However, our parallel version has two central differences: we process *all* scanlines simultaneously, and we process all edges *within* each scanline simultaneously.

The basic idea for processing each scanline is to assign to each edge on each layer a *direction*, which can be used to determine the opaque and transparent regions *between* edges on a scanline. When crossing a *forward* edge from top to bottom on a scanline, we move from a transparent region to an opaque region (i.e., from outside a polygon to inside), and vice versa for a *backward* edge. For each layer we compute a set of *counters*, one for each region between edges, with the interpretation that non-zero counter values for layer M indicate opaque regions of M. At the top of each scanline, the counters are 0. Counter values can be computed sequentially, top to bottom through the scanline, as follows: when crossing a forward edge, increment the counter for the following region; when crossing a backward edge, decrement the counter. In this fashion, even overlapping opaque regions are correctly accounted for. Boolean operations on regions are then transformed into Boolean operations on counter values, e.g., layers M_1 and M_2 intersect in a particular region if both have non-zero counter values in this region. The edges that bound these regions can be identified by looking for where Boolean counter combinations change across edges. Similar ideas can be used to infer where vertical boundary edges must exist.

Surprisingly enough, these counter values can also be computed in parallel. Assume edges in each scanline are indexed in increasing order from 0 at the top of each scanline. With each edge k in each scanline, we associate an increment I_k: *+1* for forward edges, *-1* for backward edges for each layer. Then the effect of sequentially traversing the edges, incrementing and decrementing, simply produces a counter value of $\sum_{j=0}^{k} I_j$ for the region below edge k. However, this sum can be computed in parallel using a **plus-scan** operator on the list of I_j data associated with each scanline. Note that all scanlines are being processed in parallel. Recall that the entire edge set exists as a single large linear list; segments of this list represent individual scanlines, the ends of which can be determined simply finding those edges whose minimum X coordinates differ from their neighbors.

The overall complexity of a complete Boolean operation is $O(\log N)$, determined as follows. The operations of finding the ends of each scanline, computing Boolean combinations of final counter values, and determining which edges bound output regions, each require constant $O(1)$ time. The parallel **plus-scan** operations require $O(\log E)$ time, where E is the length of the longest individual scanline list. However, since E has expected size $O(N^{1/2})$ for N total edges, we have $O(\log N^{1/2}) = O(\log N)$ time for the scan operations. Hence, the total time complexity for a Boolean operation is $O(\log N)$.

3.4. Region Numbering

Region numbering is performed to assign electrical net numbers to each electrically connected region; region numbering assigns the same number to each edge of the same connected region of a set of polygons. A good serial example is the scanline algorithm due to Szymanski [15], which uses $O(N^{1/2})$ space for N edges, and requires no global topology information during the scanline motion. This region numbering process has two major phases: a *preliminary* phase to assign temporary region numbers to each edge, and produce control information for later region merging; and a *renaming* phase to assign final, unique region numbers based on the results of the first phase.

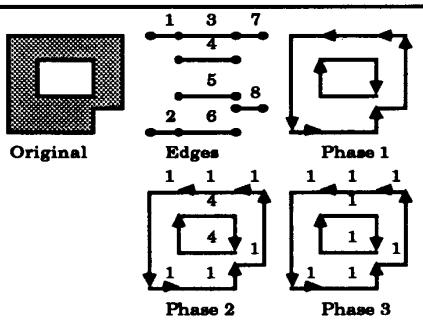

Figure 2. Region Numbering

Our algorithm for region labeling also works in several phases. The first phase determines local edge connectivity through the use of Boolean operations, and constructs a circular pointer list through all the edges forming the perimeter of each connected object. The second phase determines which processor ID is the smallest among all the edges that form one connected list and propagates it to each edge on the connected object; this ID number becomes the unique region number. The third and final phase determines the presence of any holes in the connected region and marks that they are connected to the perimeter of the region in which they are embedded. This process is illustrated in Fig. 2. A key point here is that *all* such regions in the mask are found in parallel.

The first phase begins by re-sorting the scanline data so that adjacent pairs of scanlines are tagged. This is required because, strictly speaking, edges only exist *between* scanlines. Since our aim is to deduce where vertically adjacent edges are connected, we must essentially look on both the left and right of each scanline, which can be accomplished by examining scanlines in pairs (hence the need for the tagging). A sorting operation turns out to be a simple way of computing these pair tags, with time complexity $O(\log^2 N)$ for N edges. Boolean counter operations then take place on each scanline pair to identify edges connected by (implied) vertical edges. These Boolean operations each have complexity $O(\log E)$, where E is the largest number of edges on any pair of scanlines, and E has expected size $O(N^{1/2})$. At the end of this phase, we have constructed, in effect, a circular list of pointers around the perimeter of each connected object.

The second phase establishes global connectivity; each edge in each perimeter list is given the smallest ID among the processors storing these edges. Each processor ID is propagated through each perimeter list using the distance doubling technique described in [13]. Each edge examines its neighbor's ID, then its neighbor's neighbor, and so forth, each time reaching an edge twice as distant. If each edge only keeps the smallest ID seen so far, in time $O(\log L)$, a list of L edges can be given a unique region number. Since all lists are processed in parallel, the total time here depends on the longest perimeter list.

The third phase labels holes in the interior of connected regions. At the end of the previous phase, holes have also been

assigned connected perimeter lists, but with region numbers different from the regions for which they represent holes. We propagate the region number for the exterior of a region to each interior hole by walking down each scanline and looking for adjacencies of hole edges with other hole edges and exterior perimeter edges. Again using scan operations, the correct region number can be attached to all interior perimeter lists in parallel in $O(\log E)$ time, where E is the number of edges in the longest scanline, and has expected size $O(N^{1/2})$ for N total edges.

3.5. Width and Space Checking

Width and space checking is performed to verify the geometric tolerances of design rules. Traditional techniques [18] use shrink and grow operators on connected regions; these techniques have expected time complexity of $O(N \log N)$ for N edges. We present a parallel distance checking algorithm that avoids shrink/grow operations, but is also based on exploiting the extreme locality of typical design rule distances.

The algorithm works in two phases. The first phase checks for distance violations detectable within each scanline; the second phase checks for violations between scanlines. We assume that the distance check follows a region numbering operation, so that spacing violations will not be detected between edges that are actually connected. In addition, we assume that scanline lists are again sorted so that vertically adjacent edges are on consecutive processors. Within a scanline, we can check for violations of vertical spacing between horizontal edges. Each edge, in parallel, simply checks the distance to the first edge below it, then the second, and so forth until there are no more edges found within the prescribed design rule distance. Since vertically adjacent edges are on consecutively numbered processors, the edge that lies k edges below the edge on processor P_i is itself on processor P_{i+k} and can be addressed directly. All edges check below themselves in parallel. For a width check, we only compare edges with the same region number; for space checking, we look for edges from different regions. The total complexity is at worst $O(D)$, where D is largest number of edges within a design rule distance of any edge. In practice, D is quite small, essentially a constant.

The second phase checks for horizontal distance violations among different scanlines. The idea is quite similar to the vertical checking within one scanline. Each edge in one *source* scanline checks itself against nearby edges in a different, horizontally displaced *target* scanline. Each source scanline first checks its eastward adjacent neighbor, then its eastward neighbor two scanlines distant, and so forth, until no edge in any target scanline is within the relevant design rule distance of any source scanline edge. Hence, there are two different kinds of subtasks here: *finding* the next horizontally displaced target scanline, and *checking* between these scanlines.

Finding the next target scanline is accomplished using the **copy-scan** operator to distribute the top and bottom edge ID value of the scanline at X_{i+1} to each edge in scanline X_i. Since individual scanline lists are stored concatenated to form a single large list, the bottom edge in the scanline at X_i is adjacent to the top edge in the scanline at X_{i+1}. Thus, with one additional **Get** operation, the bottom edge of the scanline at X_i retrieves the top and bottom pointers of the scanline at X_{i+1}. Thus, in parallel all scanlines find their neighboring scanline in time $O(\log E)$, where E is the longest scanline, itself of expected length $O(N^{1/2})$. By appropriately managing bottom-edge pointers, it is possible using another cycle of **copy-scan** and **Get** operations for scanline X_i to get a pointer to scanline X_{i+2} and so forth. This is the mechanism by which each scanline probes out horizontally toward its increasingly distant neighbors.

Within each source scanline, each edge is numbered starting at the top of the scanline. A simple heuristic is then used so that each source edge finds a target edge around which to explore for that region of target edges within critical design rule distances. For example, a source edge about 1/3 of the way from the top of the source scanline grabs an edge about 1/3 of the way down in the target scanline. This works well when few new features appear from scanline to scanline. In any event, each source edge then uses distance doubling techniques [13] to probe up and down in the target scanline to bound the region of edges it must examine. When this region is determined, each edge in each source scanline beings sequentially examining each target edge in its bounded region for tolerance violations.

A rigorous complexity analysis for this phase is difficult. Rather informally however, the tasks of finding the top and bottom of target scanlines, predicting a starting target edge for each source edge, and searching for the relevant region of target edges all require no more than $O(\log E)$ time, where E is the length of the longest scanline. The sequential task of examining (in parallel for each source edge) each edge in each target region is $O(D)$, where D is the largest number of edges within a worst-case design rule distance of a source edge, and is in practice a small constant. In addition, this entire task must be repeated over several scanlines at increasing distance, where again we expect that $O(D)$ scanlines are within a design rule distance At the very worst, i.e., if every source edge must examine every edge in every other scanline, this is $O(N \log N)$ for N edges. This is highly unlikely, however, and our expectation is that the time is closer to $O(\log N)$ for reasonable design rules, i.e., when $D = O(1)$.

4. Connection Machine Implementation

The mask verification primitives for complete intersection, Boolean combination, region numbering and width/space checking have been implemented as 5000 lines of C language code with embedded calls to the assembly language for the Connection Machine, called PARIS [19]. Although we have already described many of the details of the parallel implementation of each algorithm in the previous sections, we treat here two general issues that affect the implementations globally.

The first issue is the representation for individual edges. Since we have restricted ourselves to Manhattan geometry for this implementation, 32 bit integers suffice for edge coordinates. Each edge is currently allocated 512 bits of storage: 256 bits for the actual edge data, plus 256 bits of swap space used in the many sorting and re-sorting operations. Even with this fairly liberal space allocation per edge, a 16K processor machine, with 64 Kbits per node, can store over 2 million edges. A full configuration, with 64K processors, can store over 8 million edges.

The second issue concerns how edges are physically mapped onto available processors. Although the Connection Machine router would allow us to store edges in arbitrary locations, some performance gain accrues to a more intelligent placement. However, we have strived not to be tied too closely to the current physical implementation of the router, in order to provide some insulation from hardware and software changes as the machine evolves. Hence, we adopt a weak assumption about locality among processors: we assume that if the numerical IDs of two processors are close, the processors are physically close. In the current hardware, this turns out to be a reasonable, mostly true assumption. Hence, when we sort into particular scanline lists, adjacent edges are bound to consecutively numbered processors. This has two advan-

tages: it allows use of the powerful **scan** operators, and it tends to reduce communication when edges need to communicate with their neighbors.

5. Performance Evaluation

This section evaluates the performance of our parallel algorithms and compares them against traditional serial algorithms. Measurements were made on a 16K-processor CM-2 machine. Ideally, we would like to compare total speedup offered by such a parallel machine with respect to the *complete* verification task. That is, we should measure the time, start to finish, for a complete mask verification, involving an entire suite of checking subtasks, format changes, file IO, off-line operations, etc., for each machine. However, since our implementations are preliminary, we do not have sufficient parallel primitives for such a test. Hence, we adopt instead a representative sequence of checking tasks as a benchmark against which to compare performance. This sequence is illustrated in Fig. 3.

For our experiments, we compare a sequence beginning with the sorting of the flattened edge lists, followed by one Boolean operation and a region numbering operation. (The parallel width/space checking implementation is currently incomplete, and we also do not currently have serial width/space checking code to compare it with.) The serial algorithms are based on those of [16,17], and are described briefly in [12], where they were also used as a baseline for comparison with dedicated mask checking hardware. There is one difference however: the current algorithms have been simplified so that they only handle Manhattan geometry, and use only integer arithmetic. This makes the serial algorithms considerably faster, and makes the comparison with our parallel implementations more fair. All serial sorting was based on an enhanced version of the standard system quicksort algorithm (UNIX^{tm} qsort), or for very large edge files, a combination of quicksort and simple merge sort techniques. These serial algorithms all have excellent time complexity. All serial code was implemented in C, and run on a VAX 11/785 under UNIX 4.3. For the serial tool suite, we measure the CPU time, which includes IO time, for the sequence of operations shown in Fig. 3. For the Connection Machine, we measure elapsed wall-clock time for the similar sequence, including IO and processing time, shown in the figure. Measurements of end-to-end time also have the advantage that they avoid the problem of vague comparability between some serial and parallel algorithms. For example, there is no single pre-processing stage for complete intersection for typical serial tools; the completely intersected form is typically derived during a Boolean computation. The serial processing sequence is typically file-based, each checking task reading from one file and writing to another. In contrast, on the Connection Machine all edges reside within the machine until all processing is completed.

For comparison against the serial tools, we employed a very simple synthetic benchmark consisting of an array of repeated crosses, where each cross is a pair of intersecting rectangles. This benchmark is parameterized in the number of crosses present, so we can measure the effect of increasingly large numbers of edges on each algorithm. This seemingly simple test case nevertheless nicely exercises each algorithm: each cross produces a square after a Boolean *AND*, and region numbering assigns a unique ID to each square. Moreover, it is easy to control precisely the size of the input and output edge data sets.

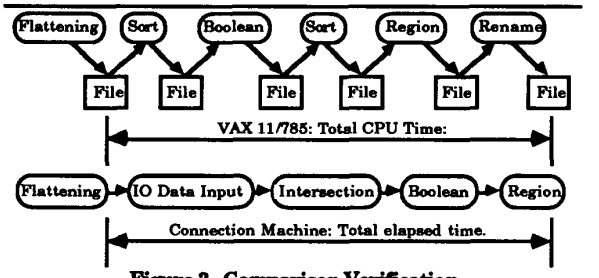

Figure 3. Comparison Verification

Figs. 4, 5 and 6 show total serial and parallel times for processing the synthetic mask benchmark, broken down by subtasks, and comparison speedup data. The repeated-cross benchmark was varied in size, measured as the number of input edges, from 200 to 256,000 edges. For extremely small masks, e.g., a few hundred edges, the overhead of IO and intersection processing completely dominate, and the speedups are poor. However, as the number of edges increases, the speedup increases to a maximum, and then begins to oscillate downward. This oscillation is very similar to the effects of finite pipeline lengths in vector supercomputers. On the Connection Machine, P processors can quickly process P edges, but P+1 edges require essentially as much time as 2P edges, since each physical processor has one virtual processor sharing it. The extra cycles to handle the virtual processors take time, but do little work, since in this case only 1 virtual processor has edge data. Similarly, we find speedup peaking at edge sizes that are multiples of the processor count, and decreasing thereafter. The slight overall downward trend is an artifact of the current overhead to manage virtual processors. This phenomenon continues until the total number of edges is sufficiently large that the actual computation time for virtual processors tends to dominate the overhead of managing virtual processors.

Two curves in Fig. 6 measure speedup with and without the total IO time to load edge data on the Connection Machine. Roughly speaking, ignoring the IO time models the (more realistic) case when the number of primitives executed is very large, and the initial IO is more completely amortized over the entire run. Recall that unlike serial algorithms which usually require file IO to link primitive checking tasks, the entire edge set stays resident in CM-2 memory throughout the entire checking sequence, with *no* intermediate IO. Without IO, the speedups range from 89 to 244; even with IO for this rather small workload, speedups range from 40 to 53. Finally, a third curve estimates roughly the speedup for a 64K Connection Machine. We assume a simple linear speedup up to 64K processors. In other words, the speedup of the 64K processor CM-2 over the 16K processor CM-2 will double for mask data containing between 16K to 32K edges and will quadruple for mask data containing more than 32K edges.

For completeness, the incremental raw time for an early version of the space checking algorithm required about 30 seconds for 64K edges; the benchmark here was the simple pattern of of repeated squares emerging from the output of the region numbering phase. All vertical and horizontal spacing violations within a scanline were flagged.

6. Conclusions

Massively parallel algorithms for several flat, edge-based mask checking primitives have been designed and implemented on a Connection Machine. Preliminary measurements on a small machine show the potential for considerable speedup over traditional serial algorithms for the same

functional tasks. Our current research is now primarily directed toward three areas. The first is performance tuning of the current implementations, and experimentation with a wider variety of both real and synthetic mask data. The second is extension of all our algorithms to support circuit extraction. The third is migration of processing tasks still performed on a serial host machine to the Connection Machine; notable candidates include the flattening task itself, and the final error distillation task, which reduces the myriad individual instances of reported errors down to their usually few root causes.

Acknowledgments

We would like to thank Dr. Rolf Fiebrich and Steve Daniel at Thinking Machines for their strong support. This work was supported in part by Thinking Machines Corporation, and by the Semiconductor Research Corporation.

References

[1] T. Blank, M. Stefik and W. van Cleemput, "A Parallel Bit Map Processor for DA Algorithms," Proc. 18th Design Automation Conf., pp 837-845, July 1981.

[2] R. A. Rutenbar, T. N. Mudge and D. E. Atkins, " A Class of Cellular Architectures to Support Physical Design Automation," *IEEE Trans. CAD of IC's and Systems*, Oct. 1984.

[3] L. Seiler, "A Hardware Assisted Design Rule Check Architecture," Proc. 19th Design Automation Conf., pp 235-238, June 1982.

[4] S. Macomber and G. Mott, "Hardware Acceleration for Layout Verification," *VLSI Design*, pp. 18-27. July 1985.

[5] J. Marantz, "Exploiting Parallelism in VLSI CAD," Proc. ICCD, October 1986.

[6] G. E. Bier and A. R. Pleszkun, "An Algorithm for Design Rule Checking on a Multiprocessor", Proc. 22nd Design Automation Conf., pp. 299-304, June 1985.

[7] S. M. Levitin, *Mace: A Multiprocessing Approach to Circuit Extraction*, Master's Thesis, Massachusetts Institute of Technology, May 1986.

[8] W. D. Hillis, *The Connection Machine*. Cambridge, Mass: The MIT Press, 1986.

[9] D. M. Webber and A. Sangiovanni-Vincentelli, "Circuit Simulation on the Connection Machine", 24th Design Automation Conf., pp. 108-113, June 1987.

[10] A. Casotto and A. Sangiovanni-Vincentelli, "Placement of Standard Cells using Simulated Annealing on the Connection Machine," ICCAD, Nov. 1987.

[11] C. Wong and R. Fiebrich, "Simulated Annealing-Based, Circuit Placement on the Connection Machine System," Proc. ICCD, pp. 78-82, Oct. 1987.

[12] E. C. Carlson and R. A. Rutenbar,"A Scanline Data Structure Processor for VLSI Geometry Checking," *IEEE Trans. CAD of IC's and Systems*, pp. 780-794, Sept. 1987.

[13] W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms," CACM, pp. 1170-1183, Dec. 1986.

[14] P. T. Chapman and K. Clark Jr., "The Scanline Approach to Design Rules Checking: Computational Experiences," Proc. 21st Design Automation Conf., pp .235-241, June 1984.

[15] T. G. Szymanski and C. J. van Wyk, "Goalie: a Space Efficient System for VLSI Artwork Analysis," *IEEE Design and Test*, pp. 64-72, June 1985.

[16] U. Lauther," An O(nlogn) Algorithm for Boolean Mask Operations," Proc. 18th Design Automation Conf., pp 555-562, July 1981.

[17] T. G. Szymanski and C. J. van Wyk, "Space Efficient Algorithms for VLSI Artwork Analysis," Proc. 20th Design Automation Conf., pp. 734-739, June 1983.

[18] J. D. Ullman, *Computational Aspects of VLSI*, Rockville, MD: Computer Science Press, 1984.

[19] "Connection Machine Model CM-2 Technical Summary," Tech. Rep. HA87-4, Thinking Machines Corporation, April 1987.

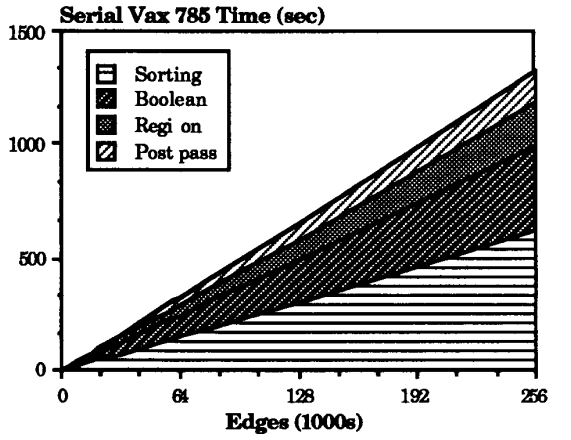

Figure 4. Serial Time

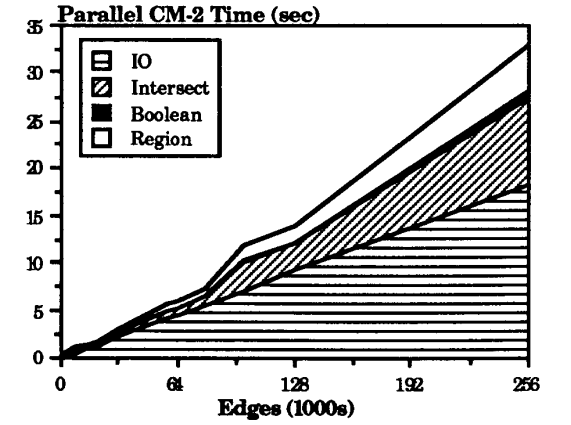

Figure 5. Parallel Time

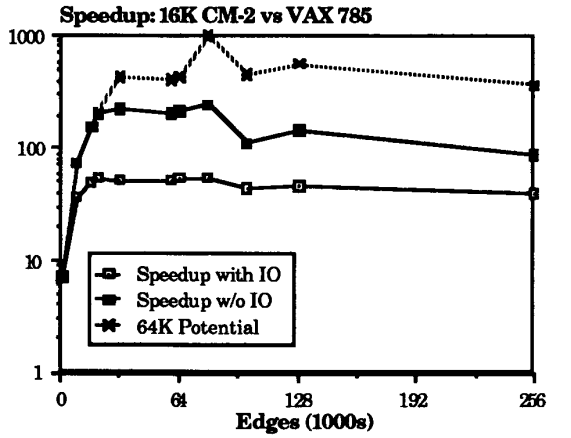

Figure 6. Speedup