CMU-CS-82-119

Computational Geometry on a Systolic Chip

Bernard Chazelle

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

April 1982

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

Table of Contents

List of Figures

Abstract

This paper describes systolic algorithms for a number of geometric problems. Implementations yielding maximal throughput are given for solving dynamic versions of convex hull, inclusion, range and inverse range search, planar point location, intersection, triangulation, and closest-point problems.

1 Introduction

The pervasive influence of VLSI in the computer science community has given research on parallel computation its second wind. In contrast with the traditional conception of parallel systems, where several computers are each assigned complicated tasks, VLSI computation, especially of systolic nature, involves the simultaneous use of a great number of very simple processors [MC,K].

As commonly referred to, systolic arrays are one- or two-dimensional arrangements of simple cells locally connected [K,K1,KL,L][1]. The essential features of systolic cells are their *simplicity*, *regularity*, and *modularity*. Performance-wise, these characteristics are definite assets, as they ensure high levels of *pipelining* and *multiprocessing*, hence providing massive parallelism. They also affect the economics of the approach by making circuit development more cost-effective. Indeed, with dropping costs of electronic components and increasing levels of circuit integration, systems designers are facing the prospect of putting hundreds of thousands of gates on a single chip, which so far constitutes a formidable challenge. Systolic architectures are one answer to this challenge. Their modularity permits the designer to decompose the system's architecture into building blocks which can be used repetitively with simple interfaces.

From the origin, the epithet *systolic* has been reserved to special-purpose devices, such as multipliers, priority queues, pattern-matchers, etc... With this perspective, systolic arrays were built with wired-in cell implementations, which was not to be a handicap as long as the overall reconfigurability of the array, an essential feature of a systolic architecture, was preserved. Thus the user was essentially given the freedom to tailor the array to the size of his problem, without having the possibility of modifying the cell definition. If one wishes, however, to optimize the cell specifications or to allow a more versatile use of the systolic device, it is essential that the cell behavior be made programmable [D]. By doing so, it becomes possible to experiment with different systolic implementations of a same scheme without having to build different chips and be caught in the bottleneck of fabrication turnaround. Also, programming the array allows the user to make it fulfill not just one function, but a whole range of related tasks. The merit of this approach partly resides in the combination *versatility & high-performance* which it affords. It must also be mentioned that it serves pedagogical purposes by putting systolic design into the hands of the laymen, thus making the conception and use of very high performance devices more accessible.

The purpose of this work is to present a *class-related* systolic processor based on the approach just described. This processor is a programmable systolic array aimed for solving a wide class of geometric problems in a highly unifying manner. This *class* of problems contains many of the most basic questions of

---

[1] The best general exposition of systolic architectures can be found in [K1].

computational geometry. Among others, we will find dynamic versions of convex hull, inclusion, range and inverse range search, planar point location, intersection, triangulation, and closest-point problems. Whenever possible, we will insist on the dynamic aspect of the problem, for it is often where systolic solutions are at their best. On the other hand, many applications areas involve problems of an inherently dynamic nature, with which we must cope. For example, air traffic control necessitates the real-time solution of closest-point problems on an ever-changing set of points.

After discussing the advantages of systolic architectures in terms of increased adaptability and cost-effectiveness, we should investigate the gains in performance to expect from a systolic treatment of computational geometry. To begin with, let us roughly describe our systolic architecture. We consider only one-dimensional arrays, i.e., arrays with a single string of cells, each connected to their one or two neighbors. Furthermore, communications with the outside world (typically, a host computer) takes place solely at either of the end-cells. It results from this configuration that although there may be full parallelism in the arrays, the number of I/O operations at any time is always bounded by a constant. We do not make this assumption for the sake of simplicity, but for the sake of *realism*. Indeed, in most applications, the systolic device will receive its data from a sequential computer, therefore the assumption we are making is not a choice but an inevitable reality.

Being now ready to turn our attention to performance considerations, we immediately derive, from the assumption above, that N pieces of data cannot be processed in fewer than N systolic steps. This may seem like a serious handicap, when compared to the $O(N^2)$ or O(Nlog N) running times typically offered by sequential geometric algorithms. One may hope at best the gain of a factor N or log N; however, asymptotic figures based on big-Oh considerations are not too relevant in the matter. Indeed, the sole performance goal in our case is to maximize the throughput, i.e., have the systolic array keep up as closely as possible with the *host/device* data rate. This data rate is dependent on the pin bandwidth of the chip, or sometimes in real-time applications, on the rate at which data is made available to the host by the outside (e.g., radar, sensor). Note that the new emphasis made here reflects yet another departure from the traditional study of computational complexity.

It is often the case that a circuit will receive streams of data, each of them pertaining to a different instance of the problem. In this case, maximizing the throughput is called *pipelining*, and to measure the adequacy of the circuit to respond to a stream of requests, we look at its *period*, a concept introduced in [VU]. Roughly, the period of a circuit is the minimum delay between two consecutive sets of inputs. Of course, it is highly desirable that our systolic designs have period O(1), which often involves preventing the occurrence of clusters or of the presence of cells waiting for others in order to complete execution. We will discuss these issues in detail later on.

In the next section, we describe the general features of the geometric systolic array, then proceed with a detailed description of the algorithms in the remaining sections. Because of the intricacy of some of these algorithms, we have chosen to keep the descriptions at a fairly intuitive level, relegating the details of the implementations as well as the proofs of correctness to the Appendix.

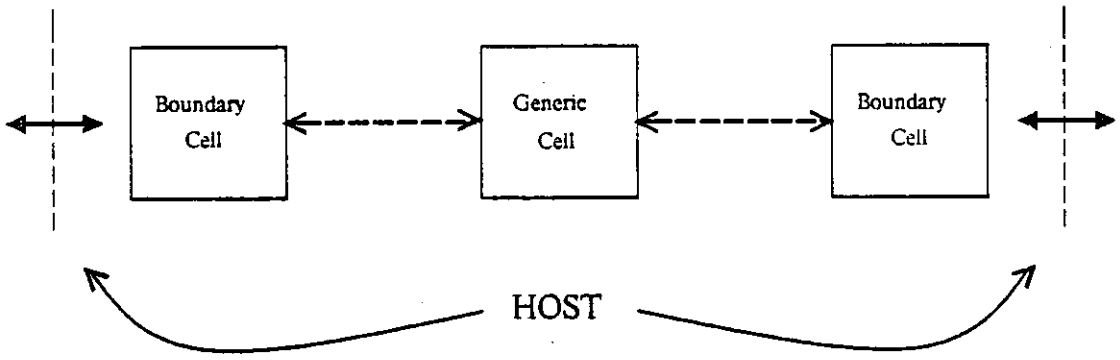

Figure 1: The one-dimensional systolic array.

2 The geometric systolic chip

Most of the systolic arrays which we will describe in this paper have the basic outlook of fig.1. Interaction with the outside world takes place solely at the end cells, called *boundary* cells. All of the other cells, called *generic*, are alike, and although boundary cells are assigned additional tasks for I/O purposes, they usually don't differ drastically from the generic cells. Each cell contains a small amount of memory, in the form of a few registers. We distinguish two kinds of registers:

1. Working registers for either storing data (point, edge, angle,...) or for providing temporary storage for the computations.

2. I/O registers for communicating data between adjacent cells.

To avoid dealing with implementation details at this point (we will take up these issues in the appendix), we may regard I/O registers as being conceptually "located" on the connection wires between the cells. These registers are protected by gates which can be either *open* or *locked* according to the current clock phase. We assume that the whole systolic array is synchronous, and that each cell operates in lock-step. For simplicity, we also assume the existence of two clocks φ_1 and φ_2 beating in opposition. This allows us to separate input and output stages easily by requiring that input (resp. output) gates should all be open (resp. locked) at φ_1 and vice versa at φ_2 (fig.2). The lapse of time between two phases φ_1 is called a *systolic cycle*. It is to be distinguished from the clock cycle internal to each cell, which is likely to be much shorter. Indeed, a systolic

cycle must correspond at least to a number of internal clock cycles necessary for a cell to complete the execution of its stored program. We should observe that this clocking arrangement is not unique; systolic arrays with asynchronous and/or adjacent cells operating in opposite cycles are perfectly feasible, so the choice made here serves only explanatory purposes, wlog.

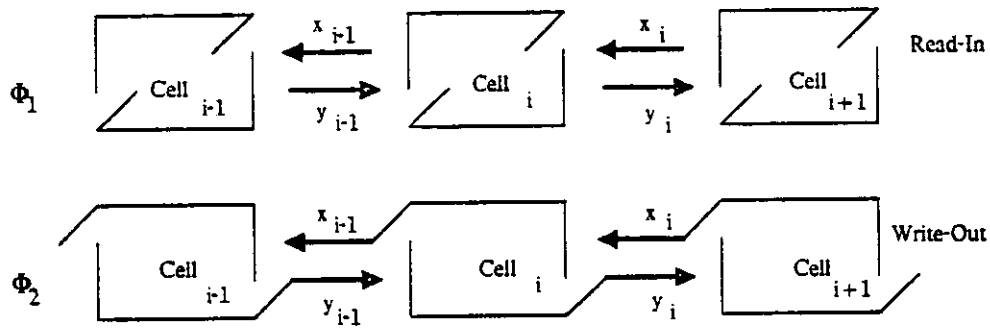

Figure 2: Handling critical paths.

The only bit of notation, used throughout, that needs be introduced here concerns the representation of points by capital letters, A,M,X,...., with a_i denoting the ith coordinate of point A in a Cartesian system of coordinates.

3 Convex hull problems

Estimating a population parameter in statistics, or simulating chemical reactions often require computing the convex hull of a set of points in a dynamic fashion [S]. In the former case, one wishes to strip away the convex hull of the set of points to remove the outliers of the sample, then remove the convex hull of the remainder, and iterate on this process until only $(1-2\alpha)N$ points remain (N and α are respectively the size of the sample and a chosen *trimming* factor). This leads to the definition of the depth of a point as the number of convex hulls that have to be stripped from the sample until the point is removed. For static and dynamic solutions to convex hull problems on a conventional machine, see [S,P1,LE,J,OV].

To fulfill our purposes, we will devise a systolic structure which supports the following operations[2].

1. Insert/delete point M.

2. Find and report all the vertices of the convex hull in clockwise or counterclockwise order.

[2]Throughout this section, we will assume the dimension of the space to be 2.

3. Determine whether an arbitrary point M lies inside or outside the convex hull.

As usual with dynamic convex hull routines, deletions and insertions proceed in very different ways. To cope with this problem, we will describe two systolic arrays, CH1 and CH2, supporting the following operations.

Array CH1

1. Insert/delete point M.

2. Report all vertices of convex hull (in arbitrary order).

3. Determine whether point M lies inside or outside the convex hull.

Array CH2

1. Insert point M.

2. Report all vertices of convex hull in clockwise (or counterclockwise) order.

3. Determine whether point M lies inside or outside the convex hull.

We observe that in order to support the operations listed at the beginning, it suffices to connect CH1 and CH2 together.

3.1 The array CH1

CH1 consists of N cells, so as to handle up to N points at any given time, each cell storing one point. All operations (updates and queries) are initiated at the input cell with the answers emanating from the output cell (fig.3).

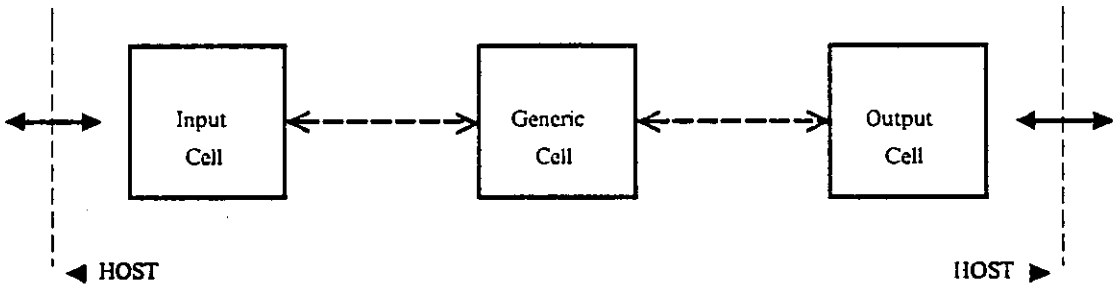

Figure 3: The overall structure of the array CH1.

Implementing Operation 1 is straightforward. Points to be inserted are pumped into the left cell, and travel

from left to right stopping at the first vacant cell. A point to be deleted is input in the same way, moving from left to right until it encounters the cell where its copy is stored, which it then marks as vacant. Note that the array does not keep track of the order of the vertices around the convex hull. Operation 3 relies on the following geometric property.

Lemma 1: Let $M_0,....,M_{N-1}$ be a list of N points in the order induced by an angular sweep around a point M. This point lies inside the convex hull of $M_0,....,M_{N-1}$ if and only if no angle of the form (M_iMM_{i+1}) [mod N] exceeds 180 degrees.

Proof: A consequence of the fact that a point lies outside the convex hull iff there exists a line containing it, with all the points on one side of the convex hull. □

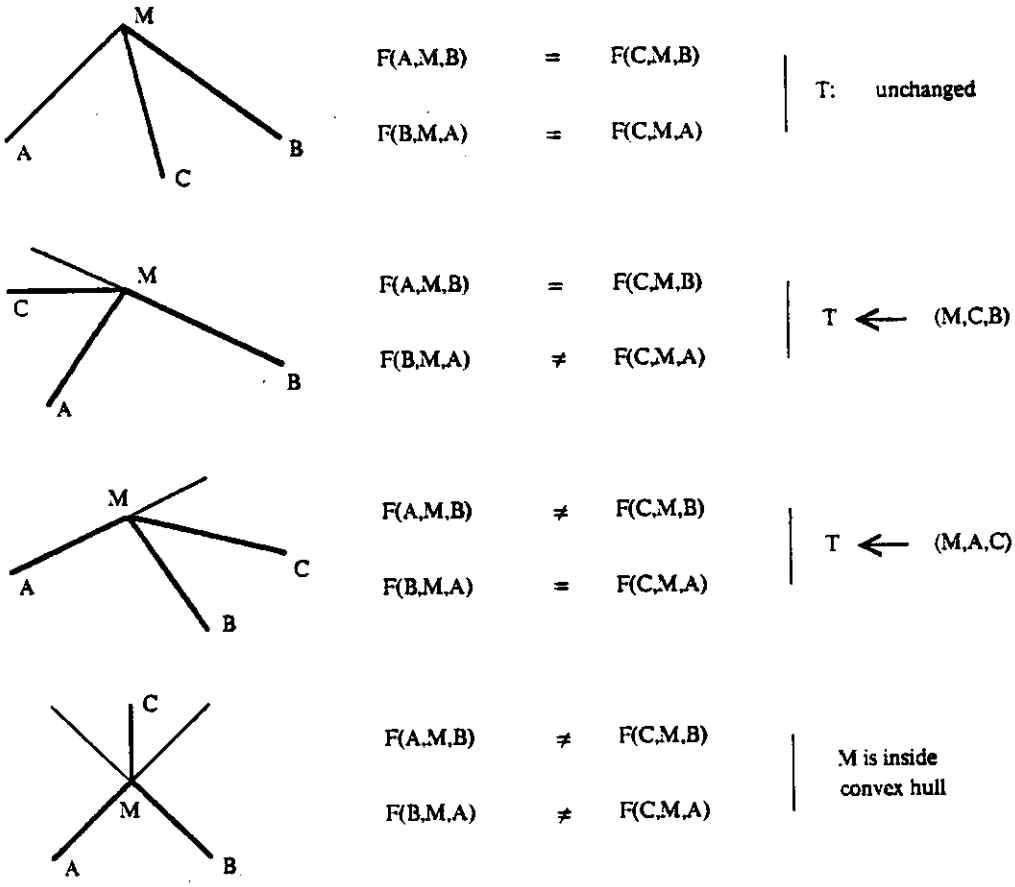

Figure 4: Testing inclusion in the convex hull.

Lemma 1 shows that we simply have to make the query point M travel from left to right, maintaining the value of the largest angle (M_iMM_{i+1}) encountered so far. This is done by a trivial case analysis, illustrated in fig.4. To alleviate the notation, we define F(M,A,B) as the sign of the expression um_1+vm_2+w, where

$uX+vY+w=0$ is an equation of the line passing through A and B[3]. This provides us with an easy characterization of whether two points M,P lie on the same side of AB, i.e., they do iff F(M,A,B)=F(P,A,B). For simplicity, we will always assume that no three points are ever collinear[4]. Let T=(M,A,B) be the triplet of points yielding the largest angle so far. M will travel along with this piece of information, which must be tested against each new point encountered, then updated before proceeding to another cell. Testing T against a new point C leads to the operations described in fig.4.

The handling of Operation 3 should be clear by now, so can proceed with Operation 2. One solution would be, in a first stage, to output copies of all the points, then in a second stage, re-input them one after the other, while executing Operation 3. To achieve the same result *in place*, we can view the systolic array as a strip of paper. The idea is then to pick it up at the input cell end and fold it over, pulling the input cell over from left to right (fig.5).

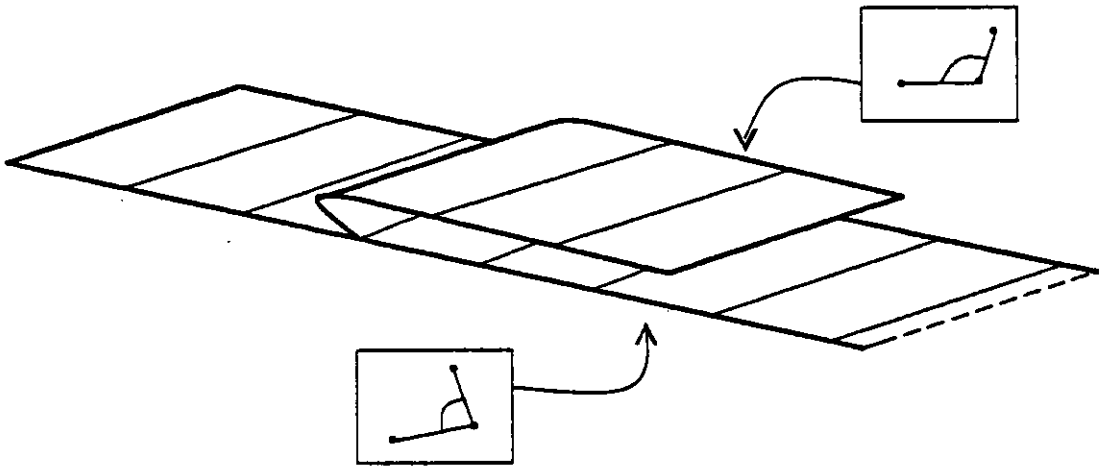

Figure 5: The fold-over operation.

To ensure that each cell will indeed look at all the others, we must update both the covering cells moving right and the covered cells not yet in motion. The updating is of the same nature as in Operation 3. Note that the left end of the folded strip will move twice as slowly as the input cell. For this reason, no operation on the systolic array should be initiated within N systolic cycles after the start of Operation 2. This will ensure that no query will ever propagate to a cell already engaged in a computation for a previous query. To implement this

---

[3] We have $u = a_2 - b_2$, $v = b_1 - a_1$, $w = a_1 b_2 - a_2 b_1$.

[4] Relaxing this requirement involves adding only a few simple, uninteresting details to the algorithms, so it is legitimate to allow such simplifications.

fold-over operation, we need essentially two signals: one is the query itself, which follows the right-end of the covering strip. The other follows the other end, and is necessary to signal the cell that at the cycle following the next, it will have to send a copy of itself to the right, thus becoming the current left front of the covering strip. See Appendix for details.

3.2 The array CH2

This structure supports only insertions, but in return, it provides an ordered description of the convex hull, at any time. Also, since the array stores only the vertices of the convex hull, it can support an arbitrary number of insertions, as long as this convex hull always keeps a number of vertices on the order of N. To begin with, let us give the geometric background behind the algorithm. Assume that $M_0,...,M_{p-1}$ are the vertices of a convex p-gon P, given in clockwise order. Let M be an arbitrary point outside P, and let Q denote the convex hull of P∪{M}. Considering the infinite line passing through an edge e of P, it is easy to see that adding M to the convex hull will cause the disappearance of e if and only if the line lies between M and P. This motivates the introduction of the function G, defined by the relation:

$$G(M,A,B) = (a_2-b_2)m_1+(b_1-a_1)m_2+a_1b_2-a_2b_1$$

Note that F(M,A,B) = sign G(M,A,B). The following result is simply a more formal statement of the remark above, and we leave out the proof - see illustration in fig.6. Once again, in the following, we shall assume that no three points may be collinear.

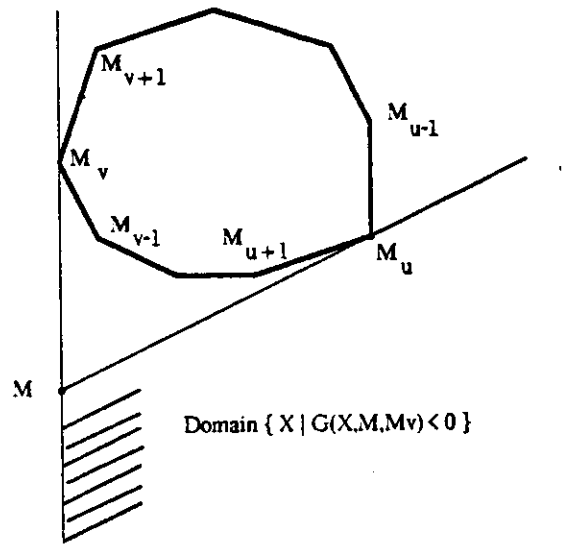

Figure 6: Computing convex hulls in clockwise order.

Lemma 2: Let $M_0,\ldots,M_{p-1}$ be the vertices of a convex p-gon P, in clockwise order. Let M be an arbitrary point and Q denote the convex hull of $P\cup\{M\}$.

1. M lies inside P iff $G(M,M_i,M_{i+1})<0$, for all i: $0\le i\le p-1$ [mod p].

2. M_iM_{i+1} is an edge of Q iff $G(M,M_i,M_{i+1})<0$. Also, if M does not lie inside P, it is a vertex of Q and its adjacent vertices are, in clockwise order, M_u and M_v, defined uniquely by $G(M_{u-1},M_u,M)<0$, $G(M_{u+1},M_u,M)<0$, $G(M_{v-1},M,M_v)<0$, and $G(M_{v+1},M,M_v)<0$.

The array **CH2** has the same overall structure as **CH1** (fig.3). Instead of a point, each cell now stores an edge of the convex hull, however, and the left-to-right order in the array corresponds to a clockwise traversal of the boundary of the convex hull. Operation 1 (*inserting point M*) causes M to travel from the input cell to the output cell, computing the function G defined above in order to determine whether M lies inside the convex hull. If it lies outside, two edges have to be added to the structure, and in general, a bunch of consecutive edges (at least one, anyhow) must be removed. More precisely, assume that $M_iM_{i+1},\ldots,M_{j-1}M_j$ are the consecutive edges of P to be removed. Upon encountering M_iM_{i+1}, M must cause the cell currently visited to substitute M_iM for M_iM_{i+1}. All the subsequent cells will delete their contents, until M encounters the first edge (M_jM_{j+1}) not to be affected by the insertion of M. At this point, the current cell must hand the cell M_jM_{j+1} to its right-hand side neighbor, and keep the edge MM_j in store. M has now ceased to cause changes in the array, and it can terminate its motion. However, there is now one cell in the array with two edges. To repair this anomaly, we make sure that the cell keeps its additional edge but forward its former contents to its right neighbor. This only causes to shift the anomaly one cell to the right, but iterating on this process will eventually cause the last non-vacant cell to release an edge to its neighbor, which solves the problem. This phenomenon is known as *rippling*, as it mimics the propagation of a wave in water. We should observe that if the last non-vacant cell has no right neighbor, *overflow* must be reported. However, the insertion may have just cause the deletion of a number of edges, in which case reporting overflow is undesirable. In general, we pose as a requirement that *no overflow should be reported if there is any vacant cell in the array*, no matter where. To comply with this rule, we must ensure that vacant cells which have edges on their right-hand side, i.e., *holes*, must be *filled* by edges from the right. To do so, it suffices to have each cell always check whether its left hand-side neighbor is vacant, in which case it must pass its contents to it. As a result, it appears that, in general, two opposite motions will take place within the array: one, to the right, corresponds to queries and insertions, while the other, leftwards, is meant to fill the holes just created. Operation 2 simply involves pumping out all the edges of the array through the input cell, thus preserving the (counterclockwise) order of the edges. Operation 3 is a simple application of Lemma 2, similar to Operation 1, yet without altering the state of the array. The query point M travels left-to-right, checking its location with respect to each edge in turn. If M is always found to lie on the same side of the edge as the interior of the polygon, inclusion must be reported, otherwise M lies outside the convex hull. See Appendix for details.

4 Inclusion, Intersection, and Closest-point problems

We next show that many of the most common geometric problems can be solved by means of a simple unifying scheme. The underlying idea, already used in arithmetic or pattern matching [K,FK,KL], exploits the inherent suitability of systolic designs to testing each input data against the contents of each cell, in a pipeline fashion.

More precisely, let $S_1,....,S_N$ be the data stored in the array, and let $a_1,....a_j$ denote a list of queries in the order with which they arrive at the input cell: the goal is to compute for each query a_k the value of $T_k^{(N)}$, defined by the recurrence relation: $T_k^{(1)} = 0$,

$$T_k^{(i)} = F(T_k^{(i-1)}, S_i, a_k)$$

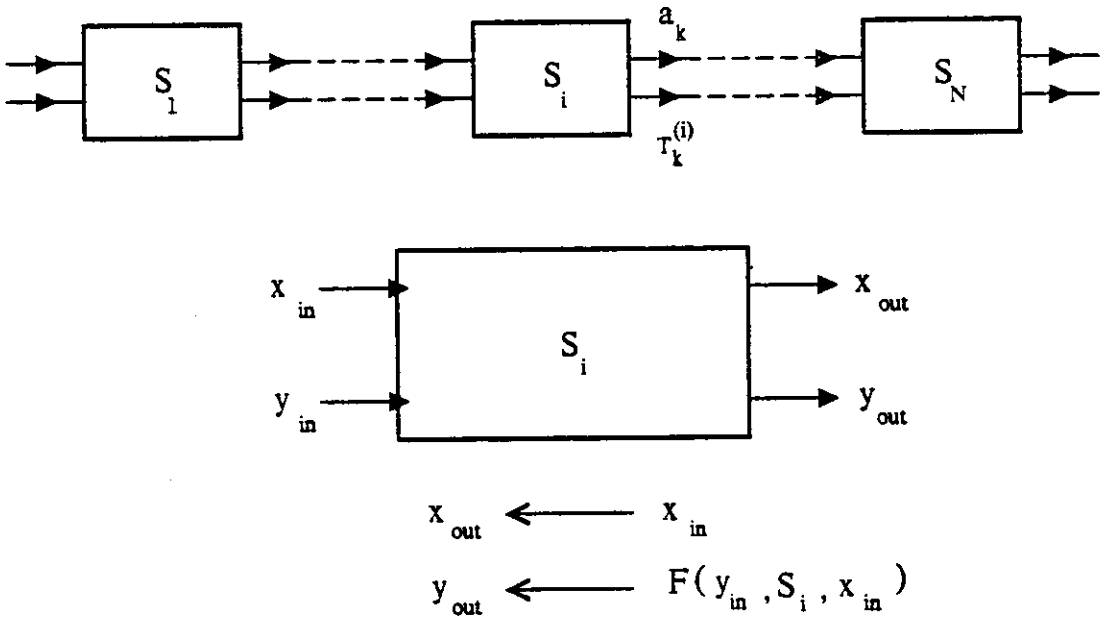

Figure 7: A systolic scheme for iterative problems.

Figure 7 sketches a systolic solution for this class of problems. As we will see, it is possible, in most cases, to make the systolic scheme dynamic, that is, capable of handling updates in the array. If no order among the S_i's is required, a *delete* (e) operation simply results in marking the cell storing e vacant, while *insert* (e) causes the storing of e in the first cell vacant from the left. If on the other hand, some order is to be preserved among the S_i's, an *insert* operation will involve searching for the appropriate (non-necessarily vacant) cell, and store the new element in it, thus possibly causing the remaining cells to ripple to the right. Symmetrically, deleting an element will incur the creation of a hole and the start of a leftward motion aimed at filling it, resulting in the propagation of the hole to the right end of the array.

For a list of applications areas where the geometric problems addressed next arise in practice, Shamos' thesis [S] is the first source to turn to.

4.1 Inclusion problems

1) Point / Polygon

Does point M lie in polygon $P = M_1,...,M_N$?

The polygon is taken to be simple[5], but no convexity assumptions are made. It is possible to achieve unit period with the following systolic scheme. The register S_i holds the pair (M_i,M_{i+1}), where the list $M_1,...,M_N$ corresponds to a clockwise traversal of the boundary of P. The variables x and y of fig.7 are respectively the point M and the pair (uv,u'v'), where uv and u'v' are the edges of P with u,v (resp. u',v') giving the clockwise direction, such that their intersection with the vertical line L passing through M forms the smallest segment so far containing M (fig.8). Testing for the inclusion of point M involves pumping M throughout the array, from left to right, updating the pair of edges in y on the fly.

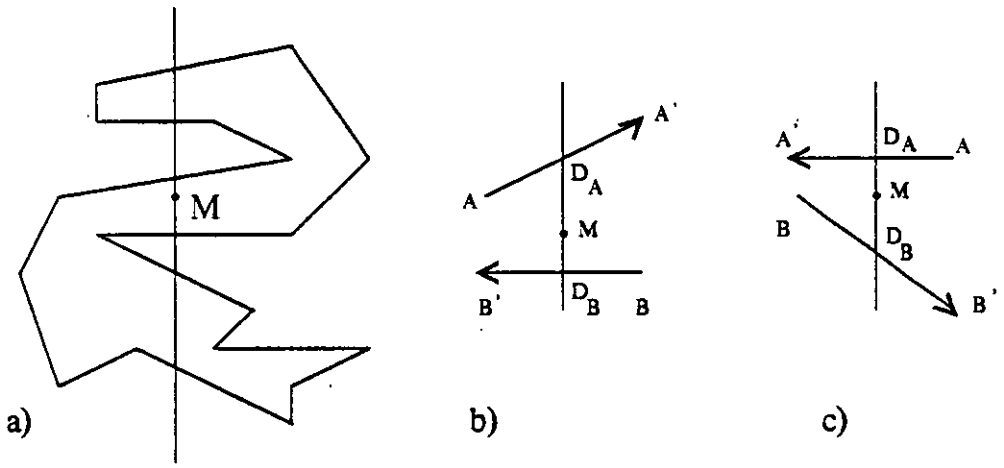

Figure 8: Testing inclusion.

[5] A polygon is simple if no pair of non-adjacent edges intersect.

Let y_{in} = (AA',BB'), with D_A = L∩AA' and D_B = L∩BB'.
Let D = M_iM_{i+1}∩L

if D lies on D_AD_B
    then
        if M lies on D_AD
            then y_{out}←(AA',M_iM_{i+1})
            else y_{out}←(BB',M_iM_{i+1})
    else
        y_{out}←y_{in}
x_{out}←x_{in}

If D_A (resp. D_B) is undefined, it can be set to +infinity (resp. −infinity), for convenience.

Eventually, the array can output an inclusion message if y_{out} falls into case b) of fig.8. or a non-inclusion signal if it falls into case c). This is a simple application of the Jordan Curve Theorem, stating that a closed curve in the plane divides the plane into two parts: the *inside* and the *outside*. Note that the scheme used above is far from unique, and other tests for inclusion may lead to equally simple systolic structures. For example, simply counting the number of intersections with the line L above and below M is sufficient, since these numbers are even iff M lies inside the polygon.

2) Planar point location

Given a planar graph with faces f_1,...,f_N and a point M, determine the face where M lies.

For this problem, several sequential algorithms with an optimal O(log N) query time exist [S,LT,P2], but for the most part, require complicated preprocessing. Instead, we can design a very simple systolic array to solve this problem with unit period. To do so, we simply represent the graph by placing in the array, next to each other, clockwise descriptions of the faces. Since in this way, each edge is represented exactly twice, and since the total number of edges of a planar graph does not exceed the number of faces, up to within a constant factor, no more than a linear number of cells will be required. We can now view the graph as a union of polygons, represented in the array by consecutive sublists of edges. Locating a query point M comes down to testing the point for inclusion with respect to each polygon in turn, as previously described, finally concluding with a report of the name of the unique polygon which contains M.

3) Range search

In one dimension, the problem consists of computing the number of segments containing a query point, given a set of N collinear segments. In two dimensions, the goal is to report the number of rectangles containing a query point, given a set of N iso-rectangles (sides parallel to the X-Y-axes) [BW,BO,NP,E,M].

The systolic array will simply store one segment (resp. rectangle) in each cell, so that the query point can scan the array left-to-right, checking for inclusion in the segment (resp. rectangle) stored in each cell, and updating the partial count. Note that the problem can be extended to arbitrary polygons instead of only iso-rectangles.

4) Inverse range search

Given a set of segments (resp. rectangles), and given a query segment (or a query rectangle), report the number of segments (resp. rectangles) that intersect the query object [BW,BO,NP,E,M].

Once again, testing pairwise intersection requires constant time, which ensures unit period. The algorithm is straightforward and needs no further development.

The last two problems arise constantly in graphics [NS], and in design-rule checking for VLSI circuits [BO,BW]. Often, however, instead of a mere number of intersections, an explicit report of all the intersecting pairs is desired. To give our systolic arrays this added capability, it is sufficient to add only a few instructions to the algorithms. One solution is to prescribe that upon encountering an intersection, a query first sends the intersecting pair forward to the next cell, then only proceeds in the same direction. Of course, this will cause a slowdown, therefore to prevent overtaking by subsequent queries, we require that before moving an object to the next cell, the algorithm first check the vacancy of that cell. To that end, each cell must keep sending *vacant* or *occupied* signals to its left hand-side neighbor. The scheme is somewhat similar to the traffic management described for CH2, so we refer to the appendix for details. We should observe that with the actual reporting of intersecting pairs, the array still yields maximal throughput, since the output flow is always kept at its maximum. The concept of period, based on *input* rate, becomes meaningless, however, since a glut due to intense output activity may cause a slowdown in the input rate.

4.2 Intersection problems

For sequential algorithms, see [S,SH,BW,BO,NP].

5) Intersection of polygons

Given two polygons P,Q, determine whether they intersect.

If we wish to determine only if the boundaries intersect, we may simply store the edges of P in the systolic array, and have those of Q travel left-to-right, testing each edge encountered for intersection. It is easy to

extend the method and solve the general problem by observing that P and Q intersect if and only if at least one of the following conditions is satisfied:

1. A vertex of P lies in Q.

2. A vertex of Q lies in P.

3. The boundaries of P and Q intersect.

Thus it suffices to add to each cell two copies of the procedure described for Problem 1); one with respect to P, the other with respect to Q. Note that each cell must check whether the passing edge is the last edge of Q, in which case, it must tag a *yes* or *no* signal to the tail of Q to acknowledge if either endpoint of the edge stored in the cell lies inside Q or not. This is straightforward, and details are left to the attention of the reader.

6) Intersection of half-planes

Given N half-planes H_1,....H_N, compute their intersection.

This problem requires Ω(Nlog N) time on a conventional machine [S.SH.B]. As usual, we expect our systolic implementation to yield maximal throughput, and thus display an overall O(N) time performance. Moreover, as we will see, it is easy to provide the array with the capability of handling queries and updates, without losing on the overall performance. This addition is very similar to the connection of **CH1** and **CH2** described earlier for the solution of dynamic convex hull problems. Actually, the similarity between the two problems is very deep, for it stems from the geometric duality which exists between convex hulls and intersections of half-spaces [B,PM].

Let I be the intersection of the N half-planes H_1,....H_N. If I is not empty, it is a convex polygon with possibly one open side, i.e., two edges that are half-lines meeting at infinity[6]. It is possible to represent I either by a list of the lines supporting the edges of I, in arbitrary order, or if we wish more information, by a list L of edges (A,B), as they appear in a clockwise traversal of the boundary. In case of an open polygon I, we require that the vertex at infinity should appear at the ends of the list. For example, we may have two points I_1, I_k, in the list

$$L = \{ (I_1,A_1),(A_1,A_2),....,(A_k,I_k) \}$$

with the understanding that the edge I_1A_1 (resp. A_kI_k) is the infinite ray starting at A_1 (resp. A_k) and passing through I_1A_1 (resp. A_kI_k).

[6]Note, for the sake of completeness, that the intersection I may also be reduced to a single half-plane or an infinite parallel strip.

Similarly to CH1 and CH2, we will design two systolic arrays INT1 and INT2 to support the following operations:

Array INT1

1. Insert/delete half-plane H.

2. Report all lines on the boundary of I, in arbitrary order.

3. Determine whether point M lies in I.

Array INT2

1. Insert half-plane H.

2. Report all vertices of I in clockwise (or counterclockwise) order.

3. Determine whether point M lies in I.

Because of the similarity with CH1 and CH2, we may only sketch the algorithms. Any standard representation of half-planes is adequate. For example, $(u,v,w,\geq)$ can be used to denote the half-plane

$$uX + vY + w \geq 0.$$

The only point to investigate about INT1 is, in Operation 2, the type of matching involved in the "fold-over" process. To begin with, it is easy to see that a half-plane H_i contributes an edge to I iff its supporting line L_i lies in the intersection of the N-1 remaining half-planes $H_1,....,H_{i-1},H_{i+1},....H_N$. Then since the intersection of L_i with the intersection of any subset of $H_1,....,H_N$ is, if not empty, a segment, a half-line, or L_i itself, it can be expressed by means of at most two points, which can then be updated as L_i is matched against each H_j in turn. All of the other features of INT1 are similar to those of CH1. As for INT2, we assume that, at all times, the array contains a clockwise description of I, with each edge stored in a separate cell. Once again, all the operations are handled as in CH2, including the hole-filling process; only the case analysis for Operation 1, the center-piece of the algorithm, needs to be detailed, which is done in the appendix.

4.3 Closest-point problems

7) Nearest-neighbor

Given N points, $M_1,....M_N$, and a query point M, determine the nearest neighbor of M - see [S,BSW,BWY].

For this problem, we allow the dimension of the space to be arbitrary and the distance to be based on any of the L_1, L_2,....L_∞ norms[7]. Whereas efficient solutions on a conventional machine involve the use of fancy

[7] Recall that the L_p-norm of a vector $(x_1,....x_d)$ in a Euclidean d-space is $[|x_1|^p + ... + |x_d|^p]^{1/p}$.

data structures (e.g., *Voronoi diagrams, planar point location search trees, k-d trees, etc...*) entailing substantial implementation overhead, a simple dynamic systolic scheme can be devised as follows:

Once again, we store one point per cell. Queries travel left-to-right, determining their nearest neighbor on the fly. To do so, each query is accompanied by the the closest point found so far. Updates in the structure are handled as in CH1, that is, inserting a point into the first available cell encountered, and deleting it by simply marking the corresponding cell vacant. If desired, a *report-all-nearest-neighbors* query can be added to the set of allowed operations. This instruction, which causes the nearest neighbor of each point in the array to be output, can be implemented by the *fold-over* procedure of CH1. See Appendix for details.

Applications areas where a device for reporting near-neighbors would be of great interest are many. Air traffic control is one example: in this situation, typically, a few radars transmit streams of signals giving updates on the position of near-by airplanes, and minimum safety distances between planes must be constantly ensured. To speed up the signaling of anomalous positions, an *emergency* output port can be reserved on each cell, with direct link to the host. Although slightly unsystolic, this feature is totally feasible as long as emergency reports remain rare events.

8) Euclidean minimum spanning tree

> *Given N points in the plane, construct a tree of minimum total length whose vertices are the given points [S].*

C. Savage, in [SA], proposes a systolic structure for computing the connected components of a graph. This structure is a one-dimensional systolic array, which can be connected to Leiserson's priority queue [L], so as to compute the minimum spanning tree in linear time.

9) Triangulation

> *Partition the convex hull of N points M_1,...,M_N into triangles, using only segments between the points.*

This problem, which arises frequently in numerical analysis (*finite element method, numerical interpolations, etc...*), has an Ω(Nlog N) lower bound on a sequential machine [S]. A one-dimensional systolic scheme can yet achieve linear time, while supporting the following features.

Array TRI

1. Insert a point in the triangulation.

2. Determine in which face of the triangulation a query point lies.

3. Report all the triangles of the triangulation by giving, for each, a clockwise order list of its vertices.

The array **TRI** computes an arbitrary triangulation, without any consideration of "goodness". Since in many cases, however, it is crucial that certain quality criteria are met, e.g, minimizing a function of the edges, the array might be used more advantageously within the framework of a more complicated heuristic. Each occupied cell may serve one of two purposes: either it stores an edge of the convex hull (R=(A,B)) with A,B giving the clockwise orientation, or it stores the vertices of a triangle in clockwise order. We also require that, from left to right, the edges stored in the cells of the first kind should appear in clockwise order (fig.9). Finally we assume the existence of a flag F to signal the first edge of either the upper or the lower chain - see description of **CH2** in the appendix for more details. With this arrangement, Operation 2 simply involves testing the query point against each triangle, carrying the containing triangle along with M, when detected (if ever), otherwise reporting an *outsideface* message, if no such triangle has been found. Yet simpler, Operation 3 involves pumping out the contents of each cell storing a triangle, one by one - see *report* operation for **CH2** in the appendix.

To handle Operation 1, two cases must be considered:

1. M lies inside a triangle (e.g., DFC in fig.9). We must replace R by, say, MCD, and insert the triangles MDF and MFC into the next two right neighbors of the current cell. This is done by rippling to the right (fig.9,10 - case 1).

2. M lies outside the convex hull, and thus will become a vertex of the new convex hull. The algorithm is very similar to **CH2**. Instead of deleting non-convex-hull edges, however, we must now insert new triangles into the array. Referring to fig.13, with AB being the edge currently examined, and C,A,B occurring in clockwise order around the convex hull, we can give the new case analysis. See example in fig.10 - case 2.

   1) Delete AB, add AM and MBA.

   2) No action.

   3) Delete AB, add MBA.

   4) Add MA.

Remark: *to read after the technical part for* **CH2** *given in the appendix.* Note that, instead of one possible *add* in **CH2** in the course of an insertion, we may now have a total of 3 *add* operations. Thus, to avoid having

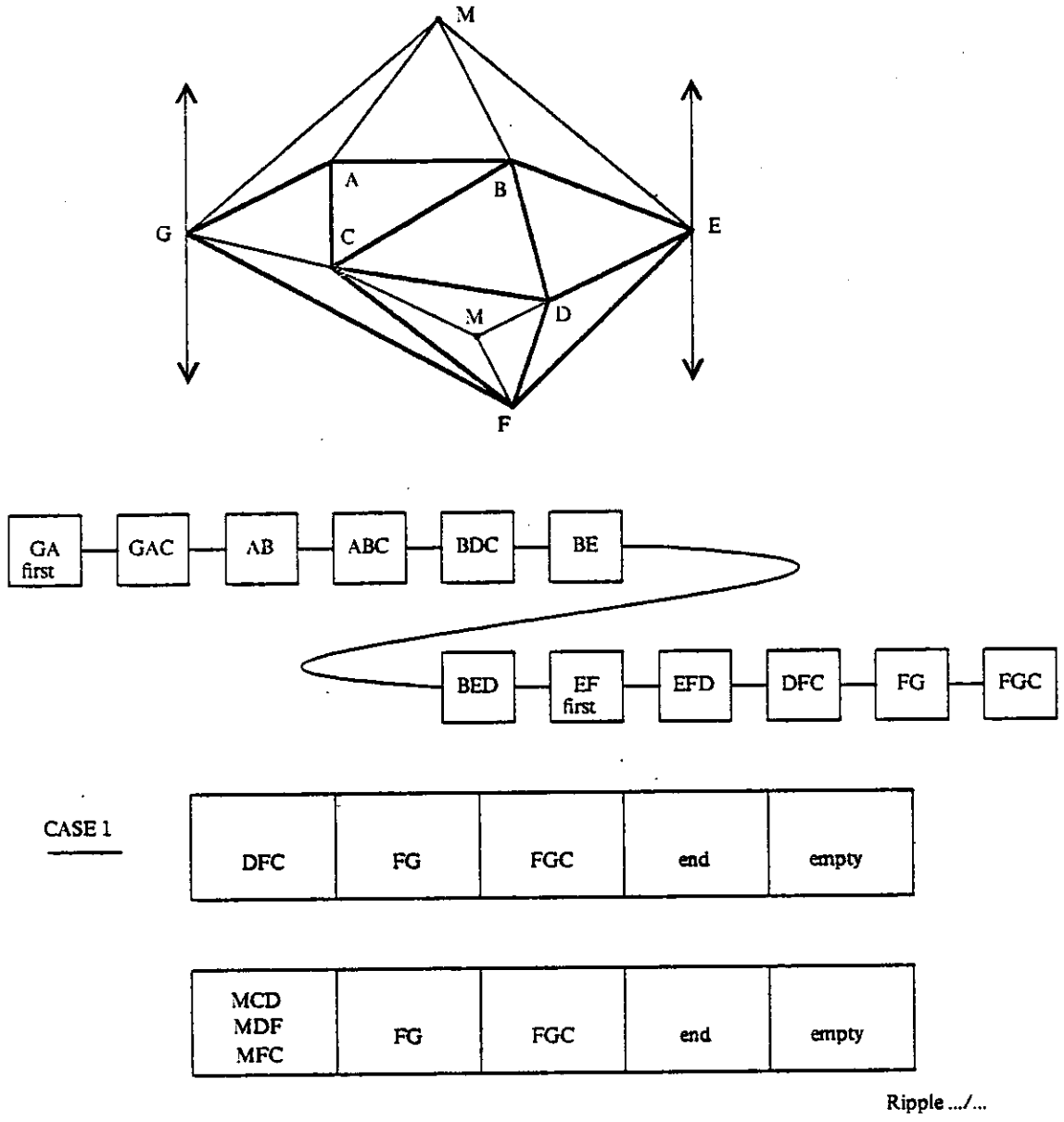

Figure 9: Continued on next page .../...

requests overtaking one another, we should add a delay of two more systolic cycles between successive requests, as compared to CH2. In consequence, a delay of 9 idle cycles between requests is certainly a safe scheduling. This margin of safety is actually overly conservative, and there is ample room for optimization.

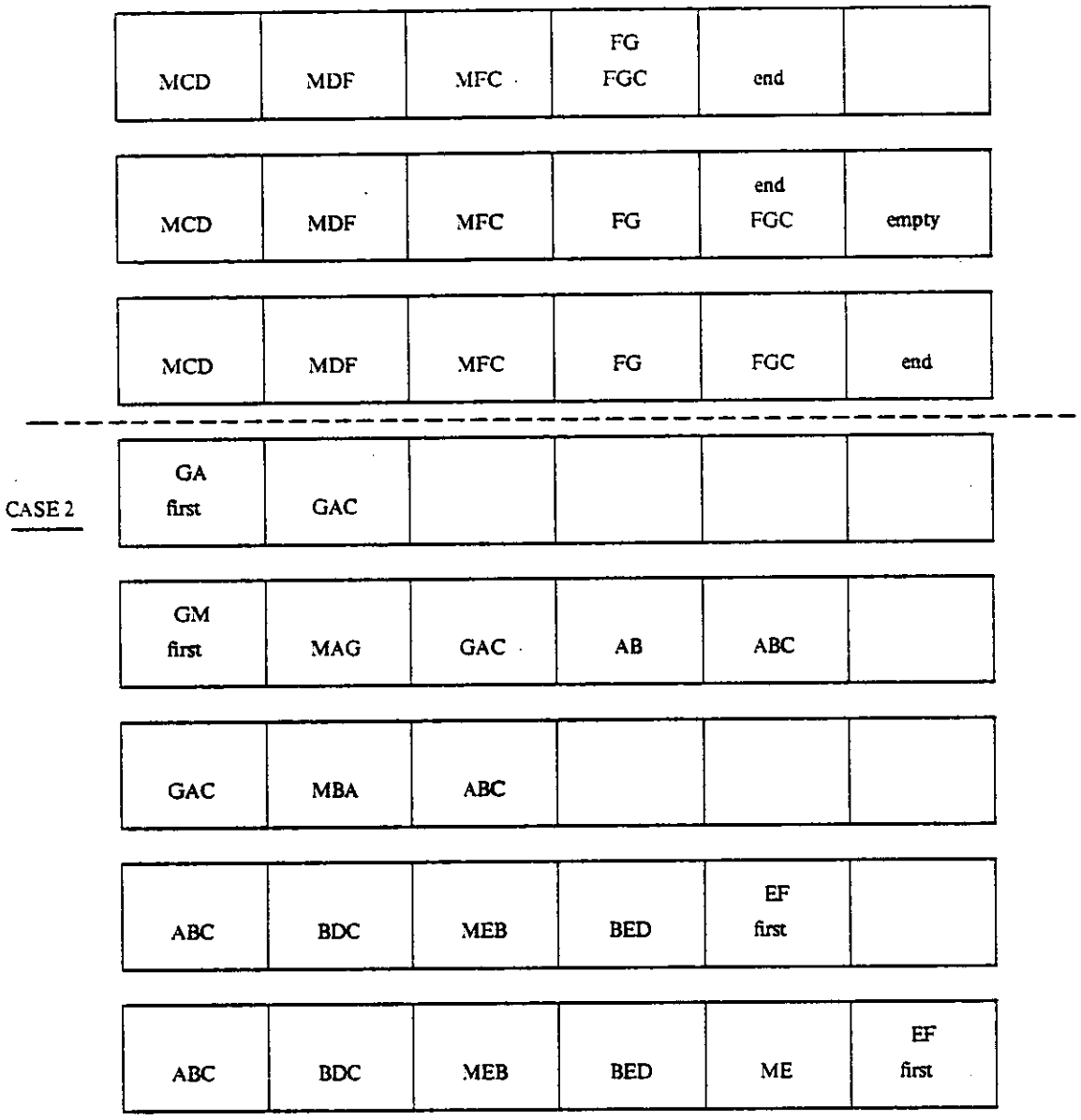

Figure 10: The triangulation array TRI in action.

5 Conclusions

The purpose of this work has been to present systolic designs for several geometric problems. Most of the algorithms described in this paper involve two distinct types of tasks. One is concerned with the actual computation of geometric functions, and is, in general, the easier to understand. The other involves initiating and granting requests, which entails moving data around, i.e., adding new items into the array or filling holes created by deletions. In general, the flow of data is irregular and not predetermined, since it is contents-dependent. With the exception of priority queues and similar structures [GL,L], this constitutes a major departure from most systolic arrays described in the literature, especially those for arithmetic computations [KL,K,FK]. Instead, most of the known systolic structures have a fixed, predetermined data flow, usually highly regular. One major difficulty with *random* motion is the absence of adequate tools for proving the correctness of the algorithms, and in particular, describing the behavior of the data flow. There certainly lie promising avenues of research.

In practice, most of the algorithms given here should undergo substantial revising before being implemented, so as to take into account the opportunities for local optimization granted by the particular applications for which the device is intended. Also, the current state of VLSI technology certainly imposes definite constraints which are bound to influence the overall design. For example, the pin/bandwidth limitation of today's chips, rightly seen by many as *the* major bottleneck, can be partly overcome by clustering several cells onto a single chip. Also, one highly desirable feature of a systolic array is that it is computation-bounded and not I/O-bounded [K1]. This amounts in practice to ensure that the cells do not spend most of the time idle, *waiting* for inputs to come. As it is, it is doubtful that this could be the case with the algorithms given here, since executing the microcode, alone, is most likely to take longer than completing any I/O operation. At any rate, it is always possible to circumvent this difficulty by providing each cell with a small random access memory (perhaps ~ 1-2K with present NMOS technology), and simulating a few tens of cells sequentially with a single processor. This solution also has the advantage of making the handling of very large inputs possible, without requiring an excessive number of chips, hence partly overcoming the inter-chip communication bottleneck. This may seem, of course, like an overt denial of the systolic philosophy, however, the presence of many cells (~ 100) within the array will largely preserve the systolic nature of the overall structure, as well as its benefits.

At the implementation level, we urge to stay away from floating-point representations, whenever possible, because of the inevitable complications which they entail. Note that in all the algorithms given above, only fixed-point additions, subtractions, and multiplications are needed, with the exception of the intersection

algorithms, which involve the solution of linear equations. In this case, division is needed, yet can be avoided, if rational numbers are kept as pairs of fixed-point numbers, as is common practice in linear programming. We should also observe that the arithmetic computations involved in the algorithms are in general very simple and limited, most of them consisting of simple fixed-point inner products.

Future work in the area of systolic algorithms includes, of course, their actual implementation and evaluation. Also, any attempt at classifying the problems that lend themselves to systolic implementations appears very worthwhile. Finally, we must once again emphasize the current need for an original description language for systolic systems, as well as new tools for studying the behavior and proving the correctness of the underlying algorithms.

Acknowledgments

I wish to thank H.T. Kung and the PSC (Programmable Systolic Chip) Group at CMU for their advice and support.

Appendix

I. The Algorithm for CH1

1. The input cell

As illustrated in fig.11, the input cell has 5 variables attached to it.

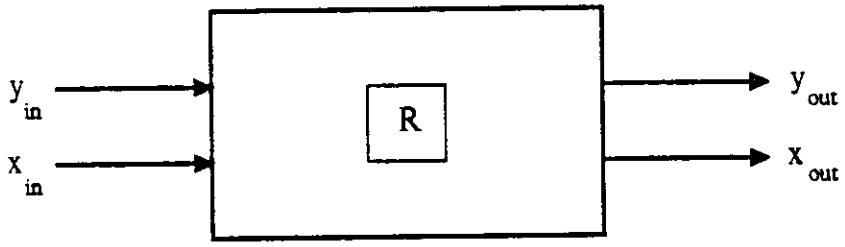

Figure 11: The input cell for CH1.

1. Variables y_{in} and y_{out} indicate the kind of operations to be performed. y_{in} and y_{out} can take on the values: *insert*, *delete*, *inclusion* (Operation 3), *report* or *repfold* (Operation 2). The purpose of this last distinction is for the consistency of the generic cell. Indeed, there must be two kinds of report signals. One (*report*) to handle the general case, the other (*repfold*) to give the additional signal that the cell receiving it is the left end of the folding strip, and therefore should pass along its own contents to its right-hand side neighbor at the next systolic cycle.

2. x_{in} can hold either the coordinates of a point to insert, delete, or test for inclusion, or have an arbitrary value when $y_{in} = report$. x_{out} serves the same function; however, when $y_{in} = inclusion$, x_{out} must hold both the query point M and the point R, for future setting of the triplet T. For simplicity, we will represent x_{out} as (M,R,0). Similarly, when $y_{in} = report$, we have $x_{out} \leftarrow (R,0,0)$.

3. R is a register with the coordinates of a point. When this point is deleted from the structure, R is marked as ε to signify that the cell is vacant.

ε is a symbol used systematically to denote an arbitrary value without significance to the computation. To shorten the description of the algorithms, we assume, throughout the paper, that all the output variables (x_{out},y_{out},....) not explicitly set to any value in the algorithm are actually set to ε.

The Algorithm

```
if y_in = insert
    then if R = ε
        then    "vacancy" R←x_in
        else    y_out←insert; x_out←x_in

if y_in = delete
    then if R = x_in
        then    R←ε
        else    y_out←delete; x_out←x_in

if y_in = inclusion
    then    y_out←inclusion; x_out←(x_in,R,0)

if y_in = report
    then    y_out←repfold; x_out←(R,0,0)
```

2. The generic cell

The generic cell is similar to the input cell, with a few addendas. In particular, it requires two more registers T and C, along with R. As explained above, as points pass over a generic cell in the *report* mode, the cell maintains a triplet of points to know its own status with respect to the convex hull of the passing points, hence the role of register T. T is a pair of points (G,H), so that the triplet is actually (R,G,H). When y_{in} = *report* or *repfold*, it is clear that y_{out} should be set to *report*. However, in the latter case, the cell must know that it must send its contents at the next systolic cycle. For this reason, y_{in} = *repfold* causes the cell to set its one-bit flag C to 1, in order to remember to do so. Thus, at the next cycle, the cell will send the contents of its register R to its right-hand side neighbor, along with a *repfold* signal. Note, however, that only occupied cells do fold over.

When a cell determines that either the point it is currently storing, or the point passing by lies inside the convex hull, it sets some appropriate flag to avoid further computation. More precisely, in *inclusion*, *report*, or *repfold* mode, y_{out} is set to *thruinclusion* in the first case and *thrureport* in the two others, so as to notice forthcoming cells to abstain from any unnecessary work. Similarly, if this happens with respect to a cell which has not started moving, T is set to *nonconv*.

The Algorithm

```
if y_in = insert
    then if R = ε
        then    "vacancy"
             R←x_in
        else    y_out←insert
             x_out←x_in

if y_in = delete
    then if R = x_in
        then    R←ε
        else    y_out←delete
             x_out←x_in

if R = ε
    then    "empty cell - pass along"
         y_out←y_in
         x_out←x_in
         stop

if y_in = inclusion
    then    Let x_in = (M,A,B)
            P = [F(A,M,B)=F(R,M,B)]
            Q = [F(B,M,A)=F(R,M,A)]
         if P∧Q
            then
                 y_out←inclusion
                 x_out←x_in
         if ¬P∧Q
            then
                 y_out←inclusion
                 x_out←(M,A,R)
         if P∧¬Q
            then
                 y_out←inclusion
                 x_out←(M,B,R)
            else
                 y_out←thruinclusion
                 x_out←(M,inside)

if y_in = thruinclusion
    then
         y_out←y_in
         x_out←x_in

if (y_in = report)∨(y_in = repfold)
  ∨(y_in = thrureport)
then
    Let x_in = (M,A,B)
    if (A,B)=(nonconv,0)
        then "y_in = repfold"
             y_out←thrureport
             x_out←(M,0,0)
    if (y_in ≠thrureport)∧(A≠nonconv)
    then
    begin

    Let P = [F(A,M,B)=F(R,M,B)]
      Q = [F(B,M,A)=F(R,M,A)]

    if P∧Q
        then
             y_out←report
             x_out←x_in
    if ¬P∧Q
        then
             y_out←report
             x_out←(M,A,R)
    if P∧¬Q
        then
             y_out←report
             x_out←(M,B,R)
        else
             y_out←thrureport
             x_out←(M,0,0)
    end

    if T = ε
        then
             T←(M,0)
    if T=(G,0)
        then
             T←(G,M)
    if T=(G,H)
        then
        Let V = [F(G,R,H)=F(M,R,H)]
          W = [F(H,R,G)=F(M,R,G)]
        begin
             if ¬V∧W
                  then    T←(G,M)
             if V∧¬W
                  then    T←(H,M)
             if ¬V∧¬W
```

```
                then    T←nonconv
            end

if y_in = thrureport
    then
        y_out←thrureport
        x_out←(M,0,0)

if y_in = repfold
    then    C←1

if C=1
    then "By convention, y_in should be ε."
        C←0
        if (T=nonconv)
            then
                T←(nonconv,0)
        y_out←repfold
        x_out←(R,T)
```

Note that in *repfold* or *report* mode, the first two generic cells, and in *inclusion* mode, the first generic cell, do not receive a full triplet (M,A,B) as x_{in}, therefore the first two generic cells do not have to execute the part of code for checking local convexity.

3. The output cell

The output cell is basically a simplified version of the generic cell. In particular, x_{out} does not need to be a triplet when the cell is in *report* or *inclusion* mode. We still give the algorithm for the sake of completeness.

The Algorithm

```
if y_in = insert
    then
        if R=ε
            then    R←x_in
            else    y_out←overflow

if y_in = delete
    then
        if R=x_in
            then
                R←ε
            else
                y_out←nodeletion

                x_out←x_in

if y_in = inclusion
    then
        Let x_in=(M,A,B)
          P = [F(A,M,B)=F(R,M,B)]
          Q = [F(B;M,A)=F(R,M,A)]
        x_out←M
        if R=ε
            then    y_out←outside
                stop
        if ¬P∧¬Q
            then    y_out←inside
```

```
            else      y_out ← outside

if y_in = thruinclusion
    then
        y_out ← inside
        x_out ← x_in

if (y_in = report)∨(y_in = repfold)
  ∨(y_in = thrureport)
    then
        Let x_in = (M,A,B)
        if (A,B) = (nonconv,0)
            then
                y_out ← ε
        if (y_in ≠ thrureport)∧(A ≠ nonconv)
        then
        begin

        Let P = [F(A,M,B) = F(R,M,B)]
           Q = [F(B,M,A) = F(R,M,A)]
        if R = ε
            then
                y_out ← hullvertex
                x_out ← M
                stop
        if ¬P∧¬Q
            then
                y_out ← ε
            else
                y_out ← hullvertex
                x_out ← M
        end

        if R ≠ ε
        then
        if T = ε
            then      T ← (M,0)
        if T = (G,0)
            then      T ← (G,M)
        if T = (G,H)
            then Let V = [F(G,R,H) = F(M,R,H)]
                    W = [F(H,R,G) = F(M,R,G)]
                 begin
                     if ¬V∧W
                         then T ← (G,M)
                     if V∧¬W
                         then T ← (H,M)
                     if ¬V∧¬W
                         then T ← nonconv
                 end

if y_in = thrureport
    then
        y_out ← ε

if y_in = repfold
    then
        C ← 1

if C = 1    then
        C ← 0
        if (T ≠ nonconv)∧(T ≠ ε)
            then
                y_out ← hullvertex
                x_out ← R
            else
                y_out ← ε
```

II. The Algorithm for CH2

For reasons which will become apparent later on, the frequency of operations initiated on the input cell must follow the rules below:

1. After starting an operation on the input cell, wait for at least 7 idle cycles before initiating another request.

2. No operation can be initiated before Operation 2 is completely finished. A special symbol *end* will acknowledge that fact.

There is nothing magic about these figures. It is sufficient that a general relation, discussed later on, be satisfied, and actually, for the sake of simplicity, our rules have been made overly conservative. Because of its generality, we begin with a description of the generic cell.

1. The generic cell

As shown in fig.12, the generic cell can be described with 6 basic variables and 3 registers R,F,C, the former storing one edge (A,B) of the convex hull. Testing the inclusion of a point $x_{in} = M$ in the convex hull involves having y_{in} set to *thruinclusion* if non-inclusion has already been determined, or *inclusion* otherwise, in which case, computing G(M,A,B) allows us to iterate on to the next cell. The variables z_{in}, z_{out} serve a double purpose. On the one hand, if z_{in} is a pair (A,B), the cell is vacant ($R = \varepsilon$) and must be filled ($R \leftarrow z_{in}$). On the other hand, once a *report* (Operation 2) has been initiated on the input cell, the contents of each non-vacant cell will get to travel towards the input cell to be eventually output. To distinguish between these two kinds of leftward motion, one bit (*report*) is tagged to z_{in}, i.e., $z_{in} = (report,A,B)$, so that the cell knows that it must only pass this value along ($z_{out} \leftarrow (report,A,B)$). Of course if $y_{in} = report$, z_{out} is set to (*report*,R).

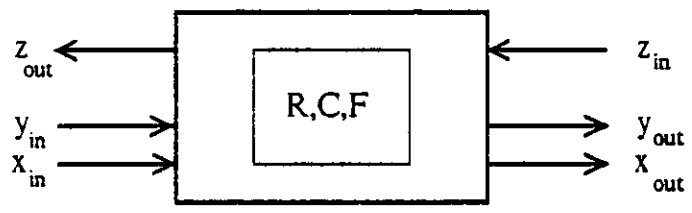

Figure 12: The generic cell for CH2.

The last case to examine (Operation 1) is by far the most delicate. To decide the status of a point M in the convex hull, Lemma 2 shows that 4 possible situations should be considered. M traveling along the array from the input to the output cell, let R=(A,B) be the edge stored at the cell currently visited, and let the variable u be set to *in* if the edge (C,A) of the previous (non-empty) cell satisfies G(M,C,A)<0, or *out* otherwise. Similarly, let v be G(M,A,B). Lemma 2 shows that the following actions should be taken.

1. u = *in*, v > 0 (fig.13.1). *Delete R to replace its contents by (A,M).*

2. u = *in*, v < 0 (fig.13.2). *No action.*

3. u = *out*, v > 0 (fig.13.3). *Delete R.*

4. u = *out*, v < 0 (fig.13.4). *Insert (M,A) before R. Send R to next cell.*

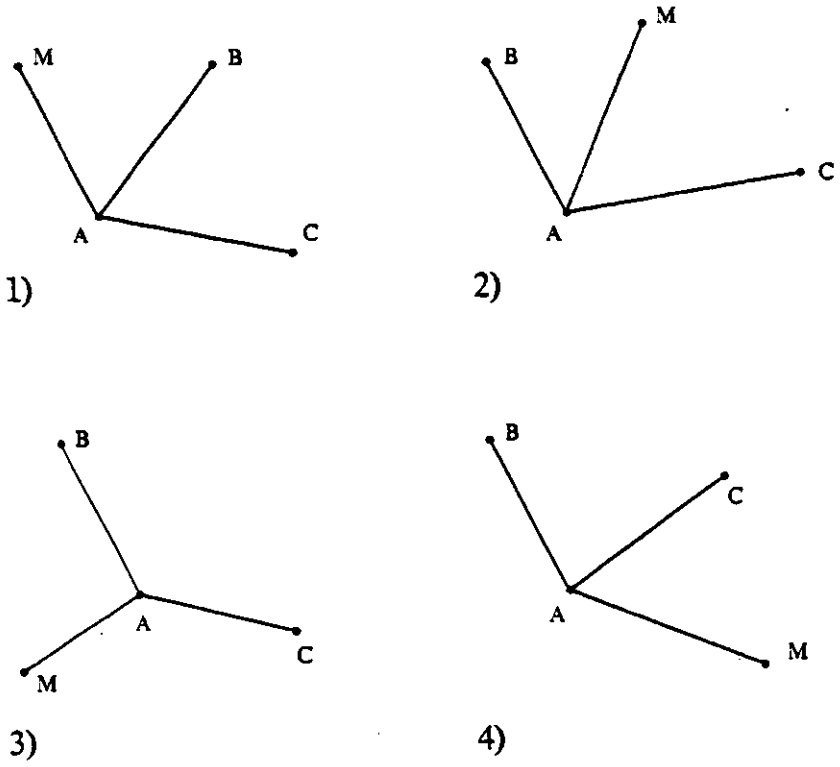

Figure 13: Establishing the status of a new point.

Note that if all 4 cases should arise, they would occur with the order 2,...,2,1,3,....3,4,2.....2 (up to circular permutation). Since we wish to pipeline the updates, it is very important that as an insert-M operation travels left-to-right, the insert signal, at any time, leaves behind the exact clockwise description of the boundary as it should be after inserting M. For this reason, we must ensure that if the 4 cases should arise, they do so in the order:

2,...,2,1,3,...,3,4,2,...,2

This problem comes from the fact that the variable **u** cannot be computed for the first cell, since it involves knowledge of the last occupied cell in the array. To overcome this difficulty, we adopt a slightly different representation of a convex polygon, which involves partitioning the boundary into two chains of consecutive edges. One, the *upper chain*, consists of the upper edges of the polygon, i.e., edges with increasing X-coordinates in clockwise order; the other, the *lower chain*, consists of the lower edges, defined as the edges pointing to the left (fig.14).

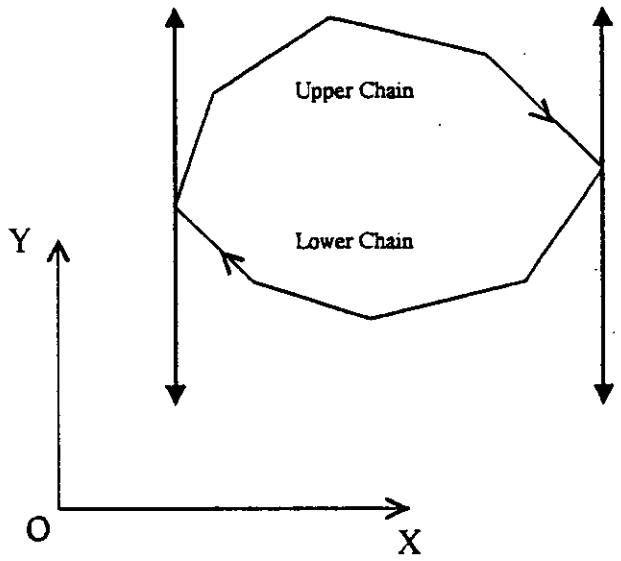

Figure 14: The partition of a convex polygon.

We now require that from left to right, the array CH2 should store first the edges of the upper chain, then the edges of the lower chain, both in clockwise order. Of course, we must assume the presence of a flag register F in each cell, which takes on the value *firstup* (resp. *firstlow*), if the cell is currently storing the first edge of the upper (resp. lower) chain. Otherwise, F is set to ϵ. The flag plays the role of **u** for the two edges in the array whose neighbors, in counterclockwise order, are conceptually two infinite vertical rays.

Situations 1 and 2 are straightforward to handle, unlike Situation 3 which creates "holes" in the array, and

situation 4 which adds one extra edge. In the latter case, the edge R and the flag F will bounce their contents on to the next cell, which will store them in its registers, and send the former contents of these registers to its neighbor. This process will iterate until the last cell ($R = end$) has been reached, thus adding one to the overall cell occupancy. While a cell is busy sending its contents to its neighbor, it must hold up the *insert* request to forward it at the next step. To do so, it uses the third register C. To handle Situation 3, i.e., to fill holes, we require that at the end of the computation, each cell checks whether it is vacant ($R = \varepsilon$), in which case it issues a *hole* signal to its right-hand neighbor ($y_{out} \leftarrow hole$), provided that y_{out} has not already been set to another value (e.g. a query/update signal). Upon receiving a *hole* signal ($y_{in} = hole$), the cell must empty its register T onto its left-hand neighbor ($z_{out} \leftarrow R$). One major difficulty is that, with a naive implementation, a right-moving query/update may miss some left-moving edges. To circumvent this pitfall, we reserve the odd systolic cycles for all leftward transfers, and the even cycles for the remaining computations.

The Algorithm

We assume that, during even cycles, all I/O variables not explicitly assigned to any value are set to ε - note that allowing this to happen during odd cycles would have disastrous effects.

Odd systolic cycles

```
if (y_in = hole)∧(z_out = ε)
    then
        z_out ← (R,F)
        R ← F ← ε
```

Even systolic cycles

```
if z_in ≠ ε
    then
        if z_in = (report,A,B)
            then    z_out ← z_in
            else    (R,F) ← z_in

if C ≠ ε
    then
        y_out ← insert
        x_out ← C

C ← ε

if y_in = insert
    then
        begin
            if (R = ε)∨(R = end)
                then
                    y_out ← insert
                    x_out ← x_in
            else
```

```
Let x_in = (M,u,w), R = (A,B),
and v = G(M,A,B)

if F = firstup
    then
        if M_1 < A_1
          then u ← out
          else u ← in

if F = firstlow
    then
        if M_1 > A_1
          then u ← out
          else u ← in

if (u = in) ∧ (v > 0)
    then
        "AM is on convex hull"
        R ← (A,M)
        y_out ← insert
        x_out ← (M,out,ε)

if (u = in) ∧ (v < 0)
    then
        y_out ← insert
        x_out ← (M,in,ε)

if (u = out) ∧ (v > 0)
    then
        "Delete R,F"
        y_out ← insert
        if (w = firstup) ∨ (F = firstup)
            then
                x_out ← (M,out,firstup)
        if (w = firstlow) ∨ (F = firstlow)
            then
                x_out ← (M,out,firstlow)
        R ← F ← ε

if (u = out) ∧ (v < 0)
    then
        "MA and AB are
        on convex hull"
        y_out ← add
        x_out ← R
        R ← (M,A)
        C ← (M,in,ε)
        if (w = firstup) ∨ (w = firstlow)
            then
                F ← w
            end

if y_in = add
    then
        y_out ← add
        x_out ← (R,F)
        (R,F) ← x_in

if y_in = report
    then
        if (R ≠ ε) ∧ (R ≠ end)
            then
                z_out ← (report,R)
                y_out ← report

if y_in = inclusion
    then
      begin
        if (R = ε) ∨ (R = end)
            then
                y_out ← inclusion
                x_out ← x_in
        else
            Let R = (A,B)
            x_out ← x_in
            if G(M,A,B) < 0
                then    y_out ← inclusion
                else    y_out ← thruinclusion
      end

if y_in = thruinclusion
    then
        y_out ← thruinclusion
        x_out ← x_in

if (R = ε) ∧ (y_out = ε)
    then
        y_out ← hole
```

2. The input and output cells

We need not give the details of the algorithms for these cells, since they are merely simplified versions of the generic cell. Before computation starts, we assume the presence of R = *end* in the input cell. For this cell, the odd cycles will be idle, and except for a special treatment for the first three points entering the array, most of the behavior of this cell is identical to that of the generic cell. As for the output cell, its most notable feature is to detect and report possible overflows, as well as outputting an inclusion message if y_{in} = *inclusion*, and a non-inclusion message if y_{in} = *thruinclusion*.

3. Correctness of the algorithm for CH2

To begin with, we should note that along with an insert-M request, two flags (u and w) should be tagged to M. The variable u is, as shown above, the status *in* or *out* of M with respect to the previous cell, and w is a flag set to *firstup* (resp. *firstlow*) if the next edge created, MA, happens to be the first of the upper (resp. lower) chain. This information is needed when the first edges are deleted by repeated occurrences of Situation 3, and w is thus the only way to acknowledge the first new edge that it is indeed the first edge of a chain. It is important to realize that filling holes with a rightward motion of edges is meant only to improve the performance of the array, i.e., put the limitation on the size of the convex hull rather than on the number of operations which can be performed. For this reason, we may first show the correctness of the algorithm when all the instructions related to that hole-filling job are dropped. This involves ignoring odd cycles as well as the last if-statement of the main algorithm. The only point remaining to be checked is that y_{out} is always set only once.

In order to do so, we may start with a few helpful observations. Let us call an even *phase* the conjunction of an even followed by an odd cycle. The rules on operations rate specified above impose a delay of at most 4 even phases between two consecutive operations. However, an *insert* operation may entail the loss of one phase, caused by the possible (unique) setting of C, thus reducing the above delay to 3. On the contrary, a cell may issue a hole signal ($y_{out} \leftarrow$ *hole*) possibly at every even phase, and similarly a cell is in a position to respond to a hole message at every odd phase ($z_{out} \leftarrow (R,F)$). From these facts, we derive in particular that whenever $C \neq \varepsilon$, we also have $y_{in} = \varepsilon$, form which it is easy to see that there is never any conflict in setting y_{out}. Now including the hole-filling instructions, we only have to show that there is no conflict in setting the register R. More precisely, we must prove that whenever $z_{in} = (\Lambda,B,F) \neq \varepsilon$, we have $R = \varepsilon$. This comes from the fact that $z_{in} = (\Lambda,B,F)$ if and only if, at the previous odd cycle, y_{out} had the value *hole*. This, in turn, implies that at the end of the previous even cycle, we had $R = \varepsilon$. Since, in addition, R can only be set to ε (if it is ever) at odd cycles, our proof is complete.

The last item to verify is what precisely motivated the distinction between odd and even cycles: the

assurance that all right-moving queries or updates encounter all the edges of the array. If $z_{in} = (A,B,F) \neq \varepsilon$, the first action taken by the cell at an even cycle is to set (R,F) to z_{in}, so that a *inclusion* or *insert* operation at that cycle will effectively deal with the just-left-moved edge. On the other hand, since the edge leaves the cell only under a $y_{in} = hole$ situation, the cell will not have to handle any query/update at the next even cycle, so it may leave the cell without missing any matching, which proves our claim. Our final investigation concerns the storage efficiency of the array. We have claimed that no overflow will ever occur, as long as the number of vertices in the convex hull, at any time, does not exceed N/2. We must now support this claim.

The above assumption clearly implies that no more than N/2 cells are occupied ($R \neq \varepsilon$) at any time, since inserting a vertex involves, first, deleting old edges, then adding the new ones. Trouble may arise, however, if edges tend to cluster towards the output cell. To dispel that worry, we introduce the concept of *leading front*, defined as the rightmost cluster of occupied cells, i.e., the rightmost group of cells without $R = \varepsilon$. A leading front can be characterized by the position H of the first cell, measured as its distance to the input cell, along with the length L of the cell. To prove the absence of leading fronts near the output cell, hence the absence of overflow, it clearly suffices to establish the following result.

Lemma 3: $H + 2L \leq N$

Proof: To look at the evolution of a leading front, suppose that the front (H,L-1) just had one cell added to it as the result of an insertion, yielding a front (H,L). From the rules, it follows that during the next 7 cycles, no more cell can be added to the front. However, a hole signal will necessarily be transmitted to the leftmost cell of the front during the first two even cycles, therefore this cell will be detached from the front by the second odd cycle, at the latest. For the same reason, a *hole* signal will reach the new leftmost cell of the front by the 4th even cycle at the latest, therefore this cell will also detach itself before the 7 cycles are elapsed, thus leaving a front (H+2,L) in the worst case, which completes the proof. □

It is easy to generalize the rules specified above, which may be useful for tuning the algorithms according to the average distribution of requests. Let A be the number of cells in the systolic array, and let α be the ratio *speed of head/ speed of tail*. If we wish to allow up to N convex hull vertices in the array, at any time, we must have the relation $\alpha A \leq A-N$, hence $\alpha \leq 1-N/A$, satisfied. On the other hand, if a (resp. b) is the delay ,measured in number of phases, imposed between consecutive *insert* (resp. *inclusion*) operations, the following relation must hold.

$$1/a \leq \alpha(1-(1/a+1/b))$$

that is,

$$1/a \leq (1-N/A)(1-(1/a+1/b)).$$

III. The Algorithm for INT2

We give only the algorithm for the *insert* operation, the others being handled in a way strictly similar to CH2. When the line L delimiting the half-plane H to be inserted intersects the current polygon I, the intersection consists (in general) of a segment VW, which must be added to the array. To do so, it suffices to tag the first intersection point encountered, V or W, along with the half-plane H, as it travels left-to-right, in order to insert VW into the array as soon as the other end-point can be computed (case 1 or 4). As usual, note the presence of the register C to buffer out the delay caused by an insertion. See fig.15.

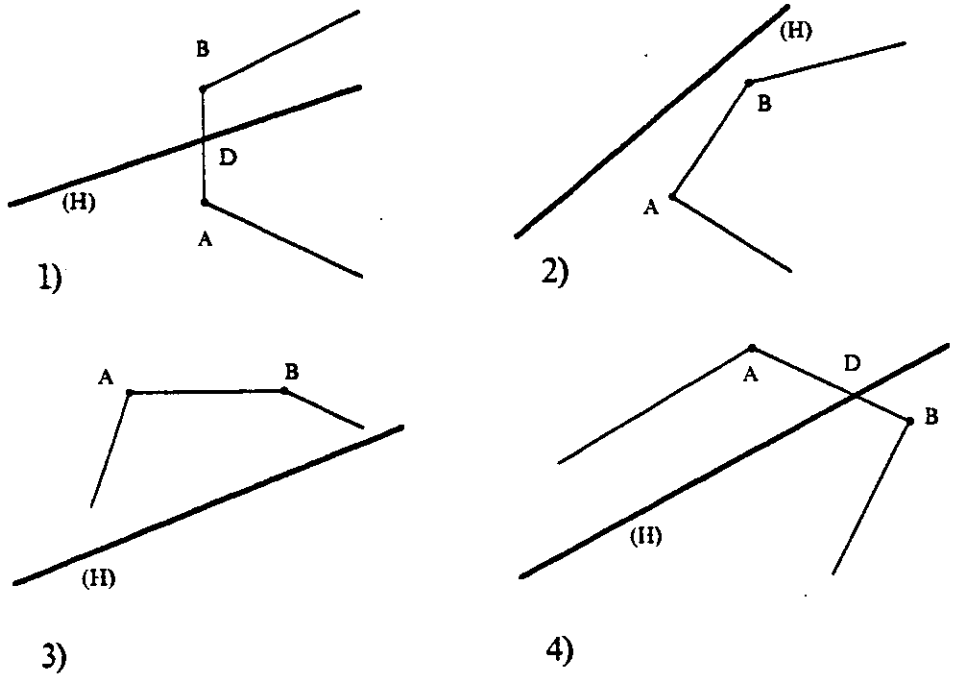

Figure 15: The various cases for INT2.

The Algorithm

```
if y_in = insert
    then
        if (R = ε)∨(R = end)
            then
                y_out ← insert
                x_out ← x_in
else
    Let R = (A,B), x_in = (H,u)
    Switch to appropriate case (fig.15).

case 1
    R ← (A,D)
    if u ≠ ε
        then
            y_out ← add
            x_out ← (D,u)
            C ← (H,ε)
        else
            y_out ← insert
            x_out ← (H,D)

case 2
    y_out ← y_in
    x_out ← x_in

case 3
    R ← ε
    y_out ← y_in
    x_out ← x_in

case 4
    if u ≠ ε
        then
            y_out ← add
            x_out ← (D,B)
            R ← (u,D)
            C ← (H,ε)
        else
            R ← (D,B)
            y_out ← insert
            x_out ← (H,D)
```

Bibliography

[BO] Bentley, J.L., Ottmann,T.

Algorithms for reporting and counting geometric intersections, IEEE Trans. Comp. C-28,9, Sept. 1979.

[BSW] Bentley, J.L., Stanat, D.F. and Williams, Jr., E.H.

The complexity of near-neighbor searching, Info. Proc. Lett. 6,6, Dec. 1977.

[BWY] Bentley, J.L., Weide, B.W. and Yao, A.C.

Optimal expected-time algorithms for closest-point problems, Proc. 16th Allertown Conf. on Communication, Control and Computing, 1978.

[BW] Bentley, J.L., Wood, D.

An optimal worst-case algorithm for reporting intersections of rectangles, Rep. CMU-CS-79-122, May 1979.

[B] Brown, K.Q.

Geometric transforms for fast geometric algorithms, PhD thesis, Carnegie-Mellon Univ., 1979.

[C] Clark, J.

A VLSI geometry processor for graphics, Lambda Magazine, Vol.1, No.2, 1980.

[D] Dohi, Y., Fisher, A., Kung, H.T. and Monier, L.

Personal communication, *PSC Group*, Carnegie-Mellon University, February 1982.

[E] Edelsbrunner, H.

A time- and space- optimal solution for the planar all-intersecting-rectangles problem, Tech. Report., Technische Universitat Graz, April 1980.

[FK] Foster, M.J., Kung, H.T.

The design of special-purpose VLSI chips, Computer Magazine, 13 (1), Jan. 1980.

[GJ] Garey, M.R., Johnson, D.S., Preparata, F.P. and Tarjan, R.E.

Triangulating a simple polygon, Info. Proc. Lett., Vol. 7(4), June 1978.

[GL] Guibas, L.J., Liang, S.M.

Systolic stacks, queues, and counters, Conf. Adv. Res. VLSI, MIT, Cambridge, Jan. 1982.

[GKT] Guibas, L.J., Kung, H.T. and Thompson, C.D.

Direct VLSI implementation of combinatorial algorithms, Proc. Caltech Conf. on VLSI, Jan. 1979.

[J] Jarvis, R.A.

On the identification of the convex hull of a finite set of points in the plane, Info. Proc. Lett. 2, 1973.

[K] Kung, H.T.

Let's design algorithms for VLSI systems, Proc. Caltech Conf. on VLSI, Jan. 1979.

[K1] Kung, H.T.

Why systolic architectures?, Tech. Report, CMU-CS-81-148, Carnegie-Mellon Univ., Nov. 1981. Also in *Computer Magazine*, Jan. 1982.

[KL] Kung, H.T., Leiserson, C.E.

Systolic arrays for VLSI, Sparse Matrix Proceedings 1978, Society for Industrial and Applied Mathematics, 1978.

[L] Leiserson, C.E.

Systolic priority queues, Proc. Caltech Conf. on VLSI, Jan. 1979.

[LE] Lee, D.T.

On finding the convex hull of a simple polygon, Tech. Report# 80-03-FC-01, Dept. of EE&CS, Northwestern Univ., 1980.

[LT] Lipton, R.J., Tarjan, R.E.

Applications of a planar separator theorem, SIAM J. Comp., 9 (3), 1980.

[M] McCreight, E.M.

Efficient algorithms for enumerating intersecting intervals and rectangles, Tech. Report, Xerox PARC, CSL-80-9, June 1980.

[MC] Mead, C., Conway, L.

Introduction to VLSI systems, Addison-Wesley, 1980.

[NS] Newman, W.M., Sproull, R.F.

Principles of interactive computer graphics, McGraw-Hill, 1973.

[NP] Nievergelt, J., Preparata, F.P.

Plane-sweeping algorithms for intersecting geometric figures, Tech. Report., Institut fuer Informatik, ETH, Zurich.

[OV] Overmars, M.H., Van Leeuwen, J.

Dynamically maintaining configurations in the plane, Proc. 12th Annual SIGACT Symp., Los Angeles, May 1980.

[P1] Preparata, F.P.

An optimal real-time algorithm for planar convex hulls, Comm. ACM, 22, 7, July 1979.

[P2] Preparata, F.P.

A new approach to planar point location, SIAM J. Comput., Vol. 10 (3), Aug. 1981.

[PM] Preparata, F.P., Muller, D.E.

Finding the intersection of a set of N half-spaces in time O(Nlog N), Theoretical Computer Science, 8,1, Feb. 1979.

[SA] Savage, C.

A systolic data structure chip for connectivity problems, CMU Conf. on VLSI Systems and Computations, Pittsburgh, Oct. 1981.

[SB] Saxe, J.B., Bentley, J.L.

Transforming static data structures to dynamic structures, Proc. 20th Annual IEEE Symp. on Foundations of Computer Science, Puerto Rico, Oct. 1979.

[S] Shamos, M.I.

Computational geometry, PhD thesis, Yale University, 1978.

[SH] Shamos, M.I., Hoey, D.

Geometric intersection problems, Proc. 17th Annual Annual IEEE Symp. on Foundations of Computer Science, 1976.

[VU] Vuillemin, J.

A combinatorial limit to the computing power of VLSI circuits, Proc. 21st Annual Symp. on Foundations of Computer Science, Syracuse, Oct. 1980.