A Simple Parallel Algorithm to Draw Cubic Graphs *

Tiziana Calamoneri† Stephan Olariu‡ Rossella Petreschi†

Abstract: The main contribution of this work is to offer a simple and cost-efficient parallel algorithm that, given an arbitrary n-vertex cubic graph G as input, produces an orthogonal grid drawing of G in $O(\log n)$ time, using n processors on a EREW PRAM. Our algorithm matches the time and cost performance of the best previously-known algorithm while improving, at the same time, the constant factors involved in two important metrics: layout area and number of bends. More importantly, however, our algorithm stands out by its conceptual simplicity and ease of implementation.

Keywords: cubic graphs, orthogonal drawing, parallel algorithms, Breadth-First Search.

1 Introduction

An *orthogonal grid drawing* of a graph is a drawing in the plane in which each vertex is mapped to a grid point and each edge is drawn as a sequence of alternating horizontal and vertical line segments running on grid lines. Recently, orthogonal drawings have attracted much attention due to their numerous applications to circuit design, data flow diagrams, entity-relationships diagrams, among many others [11]. The edges are not allowed to overlap, although a vertical segment may cross a horizontal one. In addition, the edges cannot touch vertices that are not their endpoints. If no crosses exist between any pair of edges, the drawing is called an *embedding*. A point where the drawing of an edge changes direction is called a *bend*. All vertices and bend-points lie on grid points, that is have integer coordinates.

*A major part of this work was performed while the second author was visiting the University of Rome "La Sapienza". Work supported in part by NSF grant CCR-9522093.

†Department of Computer Science, Università di Roma "La Sapienza", via Salaria 113, I-00198 Roma, ITALY

‡Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162, U.S.A.

The definition of the orthogonal grid drawing implies that graphs admitting such a drawing must have degree at most four. From among this class we only focus on *cubic* graphs, that is regular graphs of degree three, and on *at most cubic* graphs, that is graphs with maximum degree three. In spite of their apparent simplicity, cubic and at most cubic graphs turn out to be natural models for many problems arising in various applications domains. We refer the interested reader to [5, 14] where a number of applications are summarized. In the following we refer only to cubic graphs although our arguments work for at most cubic graphs, too. In fact, it is easy to see that every algorithm drawing a cubic graph works also for an at most cubic graph.

Definition 1.1 *Let k be a non-negative integer. A* k-bend drawing (embedding) *is a grid drawing (embedding) of a graph in which every edge contains at most k bends.*

A 0-bend edge is called a *straight-line edge*. The four directions on the grid with respect to each vertex are distinguished by labels from the set {*Left, Right, Up, Bottom*}. We call *free* a direction with respect to a vertex if no edge is present on it.

It is easy to see that in order to completely describe an orthogonal grid drawing of a graph, it suffices to list the coordinates of all vertices, all bends, and the directions with respect to each vertex occupied by edges.

The problem of orthogonal drawing of cubic graphs has received a good deal of attention in the recent literature. Somewhat surprisingly, the vast majority of algorithms for this problem are sequential [4, 6, 7, 12, 17, 18, 20, 21], with only few parallel algorithms reported in the literature. For example, in [16] an algorithm constructing a planar drawing with vertices placed at real coordinates is given but no bound on the area is guaranteed. The running time of this algorithm is $O(\log^2 n)$ and the number of processors required is $M(n)$, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices in $O(\log n)$ time on a CREW PRAM.

Later, this result was improved substantially: in [23] an $O(\log n)$ time algorithm, running on an EREW PRAM with $fracn \log n$ processors is proposed. The algorithm of [23] works on biconnected planar graphs but an embedding must be given as input. In [22] this result is generalized to simple planar graphs and the algorithm presented runs in $O(\log n)$ time, using $\frac{n}{\log n}$ CREW PRAM processors and returns layouts with $O(n)$ maximum edge length and $O(n^2)$ area. The number of bends is at most $2n + 4$ if the graph is biconnected, and at most $2.4n + 2$ if the graph is simple.

More recently, in [8] a first parallel algorithm dealing with non-planar, non-biconnected cubic graphs is presented. This algorithm constructs an orthogonal drawing of an n vertex cubic graph with $O(n)$ bends, $O(n)$ maximum edge length and $O(n^2)$ area in $O(\log^2 n)$ time, using n processors on a CRCW PRAM. If the drawing is 1-bend, it has at most $3/2n$ bends and has an area bounded by $(n+1) \times (n+1)$; if the drawing is 2-bend, these upper bounds are slightly better.

The main contribution of this paper is to present a new algorithm that runs in $O(\log n)$ time, using n processors on a EREW PRAM. Our algorithm returns a 1-bend orthogonal drawing of an n vertex cubic graph with $n + 1$ bends, $O(n)$ maximum edge length and $(\frac{3}{4}n + 1) \times (\frac{3}{4}n + 1)$ area. Our algorithm handles non-planar, non-biconnected graphs and neither requires an embedding of the graph as input nor computes an st-numbering (or a canonical numbering). The only input our algorithm requires is a cubic graph along with its Breadth-First tree $\mathcal{T}$ and a list of non-tree edges $\mathcal{L}$. Our algorithm relies on a novel technique based on the concept of a *solid row*. When compared with the known results in the literature, our algorithm improves the constant factors of the classical cost functions: area and number of bends. However, as we see it, our contribution lies in the structural simplicity of the algorithm.

The remainder of the paper is organized as follows: Section 2 provides a quick review of the parallel techniques used through the paper. Section 3 is devoted to a detailed description of the algorithm. In Subsection 3.1 we begin with a high level description of the algorithm; in Subsection 3.2 we provide a detailed discussion of the implementation. For illustration purposes, Section refsec4 offers a complete worked example. Section 5 contains the proof of correctness of the algorithm along with complexity results. Finally, Section 6 offers concluding remarks.

2 Basic tools

The main goal of this section is to review the basic techniques used by our algorithm. We adopt the Parallel Random Access Machine model (PRAM, for short) which consists of synchronous processors, each having access to a common memory. At each step, every processor performs the same instruction, with a number of processors masked out. In the Concurrent Read Concurrent Write (CRCW) PRAM model, several processors may simultaneously access the same memory location for both reading and writing; in a Concurrent Read Exclusive Write (CREW) PRAM model, a memory location can be simultaneously accessed by more than one processor for reading, but not for writing; in an Exclusive Read Exclusive Write (EREW) PRAM model, a memory location cannot be simultaneously accessed by more than one processor for reading or writing. The interested reader is referred to [15] for a detailed discussion of the PRAM family.

2.1 Prefix computation

Consider a sequence $a_1, a_2, \ldots, a_n$ of elements from a certain semigroup endowed with an associative operation $\circ$. The prefix computation problem asks for computing all the prefix $\circ$'s a_1, $a_1 \circ a_2$, ..., $a_1 \circ a_2 \circ \ldots \circ a_n$. The prefix computation problem is one of the fundamental techniques with applications to numerous parallel and distributed algorithms. In many contexts one is interested in prefix *sums* (that is, $\circ$ operation is addition), in prefix *maxima*, or prefix *minima* and so on.

The first optimal algorithm was devised by Cole and Vishkin [10]. It runs in $O(\log n)$ time and uses $\frac{n}{\log n}$ processors on the EREW PRAM. The basic idea of the algorithm in [10] is to partition the input sequence into $\frac{n}{\log n}$ subsequences each of length $\log n$, where, for simplicity, we assume that $\log n$ is an integer. Each of the available $\frac{n}{\log n}$ processors is assigned to one such subsequence with the mandate to compute the local prefix sums. This computation is performed in $\log n$ sequential time within each subsequence, but in parallel in all subsequences. The net effect of this operation is to have reduced the original sequence to a new sequence of size $\frac{n}{\log n}$. There is now a sufficient number of processors to solve the prefix sums problem on the new sequence in $O(\log n)$ time. Finally, the actual values of the prefix sums are calculated by a simple addition operation.

2.2 List ranking

The problem of list ranking is to determine in parallel the rank of every element in a given linked list, that is, the number of elements following it in the list. The weighted list ranking problem is similar: here, every element of a linked list has a weight and the goal is to compute, for every element of the list the sum of the weights of the elements following it in the list. Since the two problems are solved using the same technique, we shall only discuss the unweighted version.

List ranking has turned out to be one of the fundamental techniques in parallel processing, playing a crucial role in a vast array of algorithms. The simplest algorithm for list ranking running in $O(\log n)$ time and using n processors was proposed by Wyllie [24]. Although Wyllie's algorithm is not optimal, it uses an appealing technique termed *pointer doubling* or *pointer jumping*. Essentially, the idea is the following: every item in the list is initialized with a rank of 1 except for the last item whose rank is 0. The pointer doubling has for net effect splitting the original linked list into shorter and shorter lists. Specifically, in each iteration of the algorithm every list is divided into two lists of half the length. Thus, in $O(\log n)$ iterations the original list is split into n lists of length one. The ranks are updated by adding to the current rank the rank of the list element that was jumped over. The first optimal list ranking algorithm was the randomized algorithm in [3]. Later, Cole and Vishkin [10] and Anderson and Miller [3] showed that list ranking can be done optimally in $O(\log n)$ time using $O(\frac{n}{\log n})$ processors in the EREW PRAM model by producing deterministic list ranking algorithms. Both these algorithms use the same idea, namely that the original list is shortened by a factor of $\log n$ by splicing out some of the elements. Next, Wyllie's list ranking algorithm is used to compute the rank of every element in the sorter list. Finally, the removed elements are added back in the reverse order in which they were eliminated. It is important to note that when adding a removed element its rank can be computed immediately from the rank of its successor.

2.3 Euler Tour

The Euler tour technique allows one to compute a number of tree functions by reducing them to list ranking. To make our presentation self-contained, we shall present now the details of a variant of this technique. Let T be an arbitrary rooted tree with n nodes. To begin, we replace every node u in T with $d(u)$ children by $d(u)+1$ copies of u, namely $u^1, u^2, \ldots, u^{d(u)+1}$. Next, letting $w_1, w_2, \ldots, w_{d(u)}$ stand for the children of u in T, with w_i, $(1 \le i \le d(u))$, having d_i children, set for all $i = 1, 2, \ldots d(u)$:

• $link(u^i) \leftarrow w_i^1$,

• $link(w_i^{d_i+1}) \leftarrow u^{i+1}$.

Assuming that the root of T has t children, what results is a linked list $L(T)$ starting at $root(T)^1$ and ending at $root(T)^{t+1}$ with every edge of T traversed exactly once in each direction. Therefore, the total length of the resulting linked list is $2n-2$.

We can think of $L(T)$ as being stored in an array $LT(1..2n-1)$. For every node u of T all the copies $u^1, u^2, \ldots, u^{d(u)+1}$ will occur consecutively in $L(T)$. The starting position of every such group is easily determined by assigning to every node u of T a weight of $d(u)+1$ and performing a left to right prefix sum in the array T.

With every node u of T we associate $d(u)$ processors, one processor per child of u: the total number of processors used is $O(\sum_{u \in T} d(u)) = O(n)$. Note that since the children of u have been ranked, assigning these processors can be easily done. Every such processor will be responsible for setting the link pointers as we are about to describe. To begin, the starting address of the group $u^1, u^2, \ldots, u^{d(u)+1}$ in the array LT is broadcast to all the $d(u)$ processors associated with u in $O(\log\ d(u)) = O(\log\ n)$ time on a EREW PRAM. Now the processor responsible for the child w_i $(1 \leq i \leq d(u))$ of u knows the location of u^i in LT and, consequently, will set the pointer $\text{link}(u^i) \leftarrow\ {w_i}^1$ and, similarly, $\text{link}({w_i}^{d_i+1}) \leftarrow\ u^i+1$.

The above construction of the Euler tour takes $O(1)$ time using $O(n)$ processors. It is worth noting that the construction can also be obtained in $O(\log n)$ time using roughly $\frac{d(u)}{\log\ n}$ processors associated with every node u in T. Clearly, this translates into a total number of $O(\sum_{u \in T} \frac{d(u)}{\log\ n}) = O(\frac{n}{\log n})$ processors.

2.4 Tree Contraction

The problem of tree contraction involves reducing in parallel a given tree to its root by a sequence of vertex removals. Along with list ranking, tree contraction is already recognized as one of the fundamental techniques in parallel processing. It has found important applications in dynamic expression evaluation, isomorphism testing, among many others. Miller and Reif's tree contraction technique [19] uses two basic operations, RAKE - which removes all leaves in the tree, and COMPRESS - which uses a variant of list ranking to reduce the size of paths involving nodes of degree two.

These two operations, together, will cause a tree with n nodes to be reduced to a single node in $O(\log\ n)$ steps. The algorithm of [19] runs in $O(\log\ n)$ time using $O(n)$ processors in the

CRCW PRAM model.

Later, Cole and Vishkin [10] proposed an optimal tree contraction algorithm running in O(log n) time using O($\frac{n}{\log n}$) processors in the EREW PRAM model, by reducing the problem to list ranking. In spite of its optimality, however, the algorithm in [10] is rather involved. In a first stage, using the Euler tour technique they compute for every node in the tree the size of the subtree rooted at that node. In the second stage, based on this information, the tree is partitioned into a number of so-called centroid paths. The third stage involves scheduling an order for the vertex removal based on the centroid information. Finally, in the last stage the vertex removal takes place according to the schedule computed in the previous stage. More recently, Abrahamson *et al.* [1] and Gibbons and Rytter [13] proposed optimal tree contraction algorithms that are very similar in spirit to that of Cole and Vishkin, except for the way the vertices are scheduled for removal. This order is based on the initial numbering of the leaves in the original tree. First, we assume the trees represented by an unordered array with every node in the tree featuring a parent pointer along with a doubly linked list of children. restrict our attention to ordered trees in which every internal node has at least two children. We now demonstrate how such a tree T can be transformed into a full binary tree BT: if a node x has d ($d \geq 2$) children in T then, in BT, we add $d-2$ identical copies of x, namely $x_1, x_2, \ldots, x_{d-2}$ in such a way that, with x_0 standing for x,

- the parent of x_i is x_{i-1} whenever $i \geq 1$;
- the left child of x_i is the $(i+1)$-st child of x in T;
- the right child of x_i is x_{i+1} in case $i \leq d-3$, and the d-th child of x in T otherwise.

As pointed out in [1], there is no cost associated with the construction of BT, since all we need do is to reinterpret the existing pointers in T. Given a binary tree T, a *tree contraction* sequence is a sequence of trees $T = T_1, T_2, \ldots, T_m$ such that T_i is obtained from T_{i-1} by one of the following basic operations: • *prune*(v) – leaf v of T_{i-1} is removed;

• *bypass*(v) – a node v having exactly one child is removed from T_{i-1}, with the unique child of v replacing v.

To begin, the leaf e is removed by a prune operation. Next, node d having a unique child (that is f) is removed by a bypass operation. As a result, f becomes the left child of b, the parent of d in the tree.

As argued in [1], every full binary tree has an optimal contraction sequence of length O(log n), and that this sequence can be obtained in O(log n) time using O($\frac{n}{\log n}$) processors in the EREW PRAM model.

3 The algorithm

Throughout this paper we use standard graph-theoretic terminology of [2]. Only finite, simple, loopless connected graphs are considered.

The input to the algorithm is a general cubic graph (at most cubic graph) $G = (V, E)$, with $|V| = n$ and $|E| = m = 3/2n(\leq 3/2n)$, along with its Breadth-First tree $\mathcal{T}$ and a list of non-tree edges, $\mathcal{L}$. The output of the algorithm is an orthogonal grid drawing of G.

We choose to start from a Breadth-First spanning tree because all non-tree edges may connect either vertices on the same row i or vertices on adjacent rows i and $i-1$. This allows us to manage the allocation of non-tree connections locally. A non-tree edge *belongs to* level i if it *starts* from level i, that is if it connects either a vertex at level i with one at level $i-1$ or two vertices at level i.

We begin by mapping each tree-level to a different row. Let us consider a fixed tree-level i. Let n_i be the number of non-tree edges belonging to i. In order to draw these edges, we consider row i as a *solid row*, that is a block of $n_i + 1$ consecutive rows of the grid. $n_i + 1$ is called the *thickness* of solid row i.

In the following, we choose to put the root at the origin of the axes, increasing x and y towards the right and downwards, respectively.

When we say that a vertex lies on a solid row, we mean that it lies on the lowest row of the block, that is the row with the highest y-coordinate. Solid row 0 always consists of only one row.

The following two subsections describe the algorithm. Subsection 3.1 gives an overview and discusses the basic idea. Subsequently, Subsection 3.2 offers a detailed implementation.

3.1 The algorithm – a high-level description

The algorithm is divided into two phases. In the first phase the Breadth-First spanning tree $\mathcal{T}$ is drawn in order to create a "skeleton" for the entire drawing. In the second phase, the algorithm completes the drawing by working on vertices of degree one and two in $\mathcal{T}$.

1. Generate an embedding of $\mathcal{T}$ on a grid in the following way:

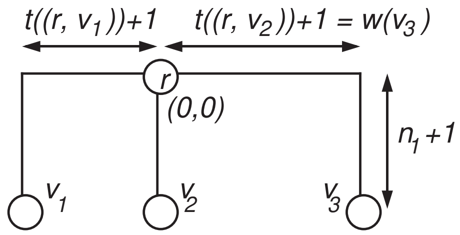

Figure 1: Embedding of the root and of its three children.

- the root is on the row 0 and all its children are on the solid row 1 occupying the left, bottom and right directions of the root. The two children on the left and right directions are connected through a 1-bend edge, while the third one (on the bottom direction) through a straight-line connection, as illustrated in figure 1.
- each vertex on a solid row i has at most two children on the solid row $i+1$. If at least one child is present, it is connected in straight-line fashion and occupies the bottom direction. The other one, if present, occupies the right direction and it is connected through a 1-bend edge.

In order to draw in parallel the whole tree we have to leave enough space in width each time we draw a new vertex v. Therefore, we introduce a function w correlated to the width of the subtree rooted at the left sibling of v, if it exists.

2. Add to the drawing non-tree edges of $\mathcal{L}$, as straight-lines, if possible; with a 1-bend edge, otherwise.

   In order to avoid that segments of different non-tree edges overlap, for each level i, we have to map each of its non-tree edges to a different grid row of solid row i. Namely, if possible, we move the endpoints of each non-tree edge so that it lies on the assigned row as a straight-line. Otherwise, we move either one or no endpoint and a 1-bend edge is introduced.

The next subsection is devoted to a detailed discussion of the two phases outlined above.

3.2 The algorithm – a detailed description

The main goal of this subsection is to flash out the details of the algorithm outlined in the previous subsection.

Let $G = (V, E)$ be an n-vertex cubic graph and let $\mathcal{T}$ be a Breadth-First spanning tree of G. We assume that the root of $\mathcal{T}$ is a vertex of degree 3. Also assume that n processors $p_1, p_2, \ldots, p_n$ are available to us on a EREW PRAM.

Before generating the embedding of $\mathcal{T}$ in the grid, we will find it convenient to compute some functions of G and $\mathcal{T}$. First, by using the Euler Tour technique, we let each vertex v know its post-order number, $p(v)$, in $\mathcal{T}$, the level $l(v)$ of $\mathcal{T}$ to which it belongs, as well as the identity of its parent, $f(v)$. In the following we identify each vertex v with its post-order number, $p(v)$.

Each tree edge $e = (u, v)$, where $l(u) = i-1$ and $l(v) = i$, is characterized by a label, $t(e)$, equal to the number of degree 3 vertices in the subtree rooted at v. $t(e)$ is obtained in the following way. We apply the parallel prefix sum algorithm to the Euler path EP associating a weigth of 1 to edges $(v, \mathrm{f}(v)), v \in V$ and degree$(v) = 3$ in $\mathcal{T}$, and a weight of 0 to all other edges. Then, $t(e)$ is equal to the difference between the prefix sums of (v, u) and (u, v).

Finally, on each level i, the number $n(i)$ of non-tree edges starting from level i is computed by applying list ranking to number the edges in $\mathcal{L}$. Each edge (u, v) in $\mathcal{L}$ chooses its orientation from lowest to highest post-order numbered vertex. Then, we construct an array A whose i-th position contains the i-th edge and the level of its starting vertex. By using one of the optimal sorting algorithms (see [9] or [15]), we sort this array in increasing order by level. Each processor p_i reads the values of levels of $A[i]$ and $A[i-1]$. Only if these two levels are different p_i becomes active. Namely, the processors becoming active are those corresponding to the first edge belonging to each level in A, that is to the edge at lowest index among all edges belonging to the same level. Each active processor p_i writes 'i' on a new array B in position corresponding to the level traced in $A[i]$. The dimension of B is equal to the number of levels in $\mathcal{T}$ plus one. The last position is set to $\frac{n}{2}+1$. The value $n(i)$ is equal to the difference between $B[i+1]$ and $B[i]$.

The reader should be able to confirm that all these labels can be computed in $O(\log n)$ time using the n processors at our disposal on an EREW PRAM.

As already mentioned in Subsection 3.1, the embedding of $\mathcal{T}$ requires to introduce a function w that represents the distance between a vertex v and its parent u in the embedding of $\mathcal{T}$, projected on the x axis.

In general, the definition of $w(v)$ is related to that v is the left or right child of $f(v)$: $w(v) = 0$ if v is left child, otherwise $w(v) = t((u, s)) + 1$, where s is the left sibling of v.

The definition of w supposes that each vertex may have at most two children. It remains to define this function for the children of the root: the leftmost child of the root has $w(v_1) = t((r, v_1)) + 1$, the middle child has $w(v_2) = 0$ and the right most one has $w(v_3) = t((r, v_2)) + 1$. Notice that the root on the origin of the axes implies a negative value for the x-coordinate of v_1.

The coordinates of the root and of its children – computed using function w – and the thickness of solid rows are illustrated in Figure 1.

The position of a general vertex v in the grid with respect to its children will be the same as r with respect to v_2 (v_2 and v_3) if v has one child (two children).

In order to calculate its position, every vertex needs to know its parent's coordinates. This can be obtained by using tree-contraction and can be computed in $O(\log n)$ time using $O(\frac{n}{\log n})$ processors on a EREW PRAM.

We note that it is possible to apply the tree contraction technique on a binary tree whose vertices have either 2 or 0 children. Then we have to transform $\mathcal{T}$, that has a root with three children and all the other vertices with at most two children.
The transformation is as follows: after giving the information of the root to its three children, we eliminate the root to compute the tree contraction on three different binary trees. The second property is guaranteed by adding a dummy vertex to each vertex having only one child.

If we sort the endpoints of a non-tree edge as we have already done to compute $t(e)$, we implicitly consider it as a directed edge from left to right. We say that the edge (u, v) is *sent* by u and is *received* by v when $u < v$. In this case, u and v are called *sender* and *receiver*, respectively. Then either $l(u) > l(v)$ or $l(u) = l(v)$ and v lies to the right of u.

The **second phase** is divided into the following steps:

Step 1. Label non-tree edges;

Step 2. Label vertices;

Step 3. Draw non-tree edges.

We now provide a detailed discussion of each of the steps above.

Step 1. First, we assign to every non-tree edge e at level i a number $c(e)$ from 1 to $n(i)$, in order to associate to each e a different row into the solid row i. This will allow us to draw all non-tree edges in parallel without conflicts in Step 3.

The details of this step follow: in each block with the same level in the array A (defined during the first phase), smaller numbering is assigned to edges with receiver at level $i-1$, following the left to right direction (that is from the smallest to the larger sender). Greater numbering is assigned to edges with receiver at level i, from left to right. As an example of c labeling, see Figure 5.b.

This step requires $O(\log n)$ time using $O(\frac{n}{\log n})$ processors on the EREW PRAM because list ranking is done in each block i of A, after ordering edges putting first edges from level i to level $i-1$ and then edges from level i to level i. Finally, each sub-block is ordered from left to right.

Step 2. The labels built in this step will be useful in Step 3. to decide the exact position of the endpoints of non-tree edges on the grid. Since we are dealing with cubic graphs, each vertex may be endpoint of at most two non-tree edges. We assign a couple of pairs to each vertex, one for each possible incident non-tree edge. Actually, for each e, we assign a pair representing the level of its receiver and its label $c(e)$. We assign $+\infty$ to all the non-defined values and then we sort the pairs inside the couples in lexicographic order. An example of such a labeling is illustrated in Figure 6.a.

This step is executed in constant time using a processor for each non-tree edge.

Step 3. At this point, each e has its label $c(e)$ traced on the labels of its endpoints. Since the first and second pairs are sequentially processed, the fact that $c(e)$ appears in the first or second pair in the two endpoints implies different possibilities to move the endpoints. Indeed, if an edge e appears in the first pair of one of its endpoints v, e can freely move v, while if e appears in the second pair of v, it is necessary to take into account that v has already been partially fixed by the edge in the first pair.

We now provide a detailed discussion of the four possible cases:

3.1. $c(e)$ appears in the first pair in both the endpoints. Then we can draw e in a straight-line fashion after moving its endpoints on row $c(e)$ of solid row i (see Figure 2.a in general and edges $(2,8),(1,4),(11,12)$ and $(9,14)$ in Figure 6 as an example).

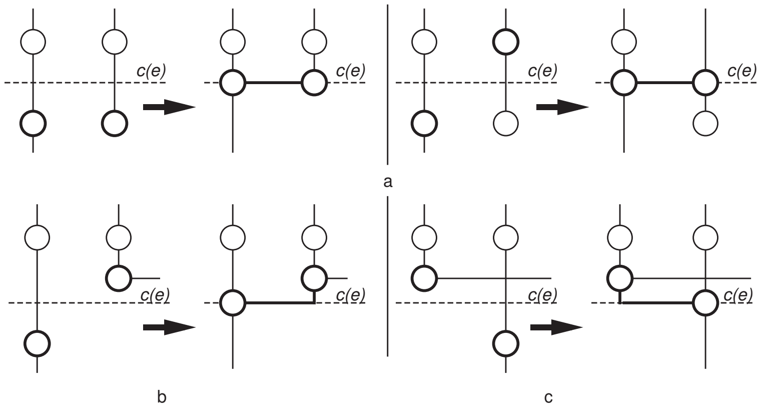

Figure 2: Different cases to draw a non-tree edge e.

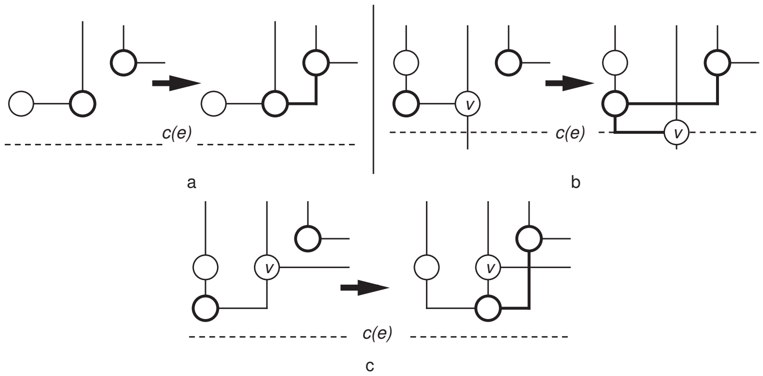

Figure 3: Different cases to draw a non-tree edge e in the situation of step 3.4 (former cases).

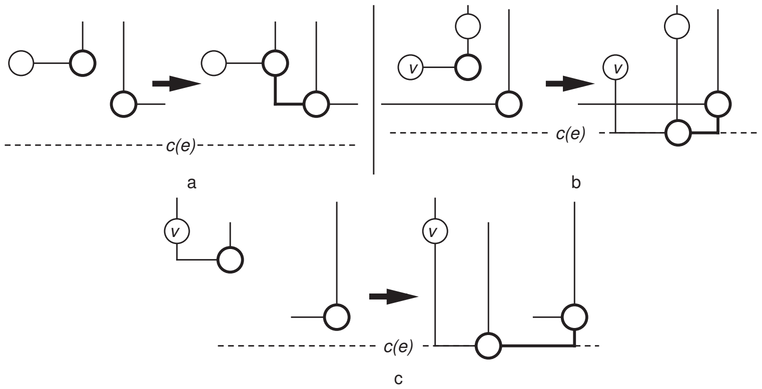

Figure 4: Different cases to draw a non-tree edge e in the situation of step 3.4 (latter cases).

3.2. $c(e)$ appears in the first pair in the sender and in the second one in the receiver. This means that the receiver has fixed its position on the grid by the non-tree edge traced in the first pair, and only the sender will be moved on row $c(e)$. e will be drawn as a 1-bend edge from the right direction of its sender to the bottom direction of its receiver (see Figure 2.b in general and edges $(5,8),(3,10)$ and $(10,12)$ in Figure 7 as an example). Observe that no conflict holds because of a chain of these edges (edges $(3,10)$ and $(10,12)$).

3.3. $c(e)$ appears in the second pair in the sender and in the first one in the receiver. This case is exactly the symmetric of case c.2: the edge is drawn with one bend from the bottom direction of the sender to the left direction of the receiver (see Figure 2.c).

3.4. $c(e)$ appears in the second pair in both the endpoints. Now, both sender and receiver have fixed the position and the bottom direction is free (see vertices 1 and 5 in Figure 7.a and Figure 3 for the general case). In this case, different possibilities can hold, depending on the situation and the relative position of the sender and the receiver.

- If the right direction of the sender is free and the sender lies on a lower row than the receiver, e is drawn as a 1-bend edge from this right direction of the sender to the bottom direction of the receiver (see Figure 3.a).
- Let us consider now the case in which the sender still lies on a lower row than the receiver but the right direction of the sender is not free (see Figure 3.b and .c in general, vertex 1 in Figure 7.a as an example); in this case, in order to draw edge e (edge $(1,5)$ in Figure 7.a), we are obliged to move on row $c(e)$ a vertex previously fixed. Let e' (edge $(1,4)$) be the edge occupying the right direction of the sender of e and v be its other endpoint. We distinguish two cases according to the fact that e' is straight-line or 1-bend. In the former case (see Figure 3.b), we move v on row $c(e)$, adding a bend to the drawing of e'. Then a 1-bend edge e is drawn from the right direction of its sender to the bottom direction of its receiver. In the other case (see Figure 3.c), we move the sender of e on the bend under v. Also in this case e is drawn as a 1-bend edge.
- We now have to consider all cases coming from the configuration of the sender on a row higher than the row of the receiver. If the left direction of the receiver is free then (symmetrically with respect to the case in Figure 3.a) e is drawn as a 1-bend

edge from the bottom direction of the sender to the left direction of the receiver (see Figure 4.a).

– If the left direction of the receiver is not free, we have some subcases depending from the edge e' incident to the sender and already drawn during a previous case (see Figure 4.b and 4.c). Observe that if e' is straight-line, then the bottom position of the sender's mate must be free, in view of the definition of label c. Therefore it is possible to bend edge e' and move the sender on row $c(e)$ (see Figure 4.b). If e' already has a bend, then we stretch e' so that the sender can reach row $c(e)$ (see Figure 4.c). In both cases, after the movement of the sender on row $c(e)$, the row where the sender lies is lower than the row of the receiver, therefore a previous case can be applied.

These four cases are sequentially considered to avoid conflicts and each of them requires $O(1)$ time on a EREW PRAM model, using a processor on each non-tree edge.

4 A worked example

Consider the cubic graph $G = (V, E)$ in Figure 5.a. Let the subgraph induced by the bold edges in G represent a Breadth-First spanning tree $\mathcal{T}$. In Figure 5.b, $\mathcal{T}$ is depicted with all the labels on its vertices, edges and levels generated during the first phase and during Step 1 of the second phase.

In Figure 6.a, $\mathcal{T}$ is embedded on the grid according to all the parameters mentioned in the algorithm and the labels on vertices consisting of couples of pairs are introduced. In Figure 6.b we execute Step 3.1. of the algorithm, that consists in embedding the non-tree edges as straight-lines. In particular, let us analize edge $(1, 4)$ whose c-label is 2; the couples associated to its endpoints are $((3, 2), (3, 3))$ and $((3, 2), \infty)$. Therefore 2 appears in the first pair of both the endpoints; hence, vertices 1 and 4 are moved to row 2 of solid row 3.

Figure 7.a shows the partial drawing after the execution of Steps 3.2. and 3.3. (the latter one is null in this case). Finally, the output of the algorithm is shown in Figure 7.b.

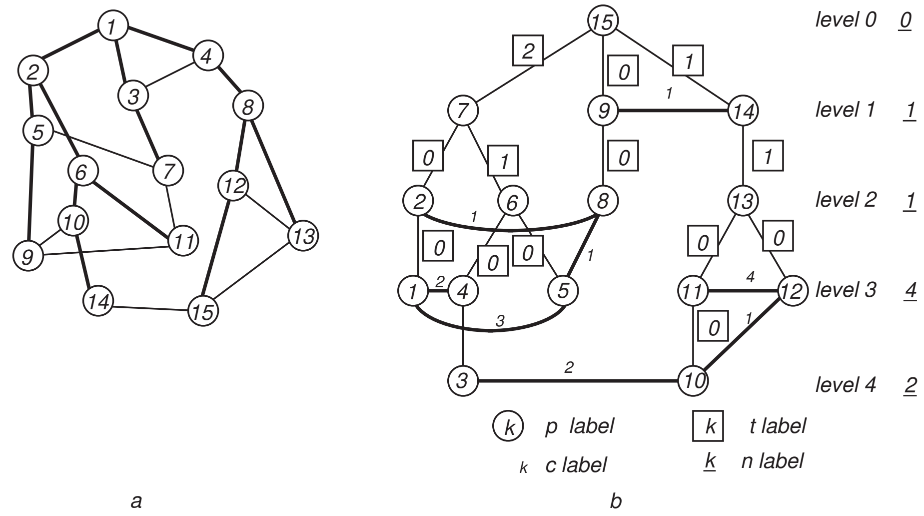

Figure 5: *An at most cubic graph G and a spanning tree $\mathcal{T}$ for G with post-order numbering.*

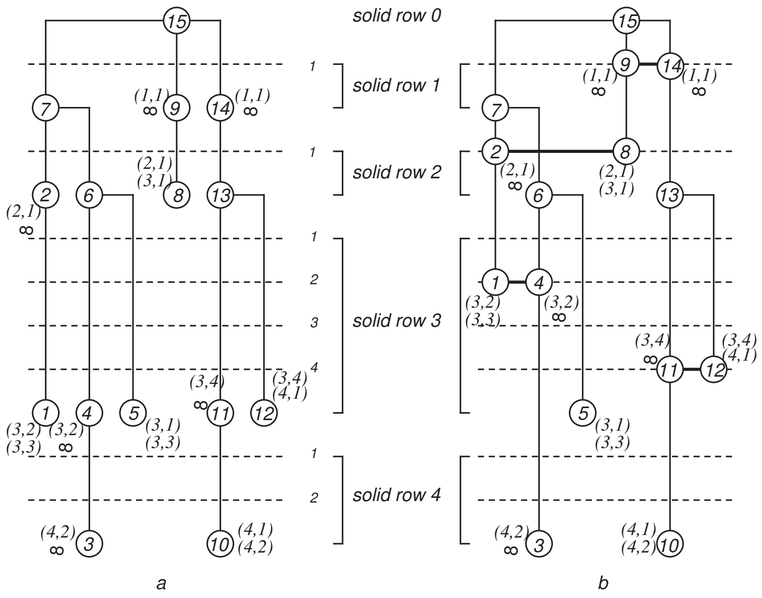

Figure 6: *The embedding of $\mathcal{T}$ and a partial drawing of G.*

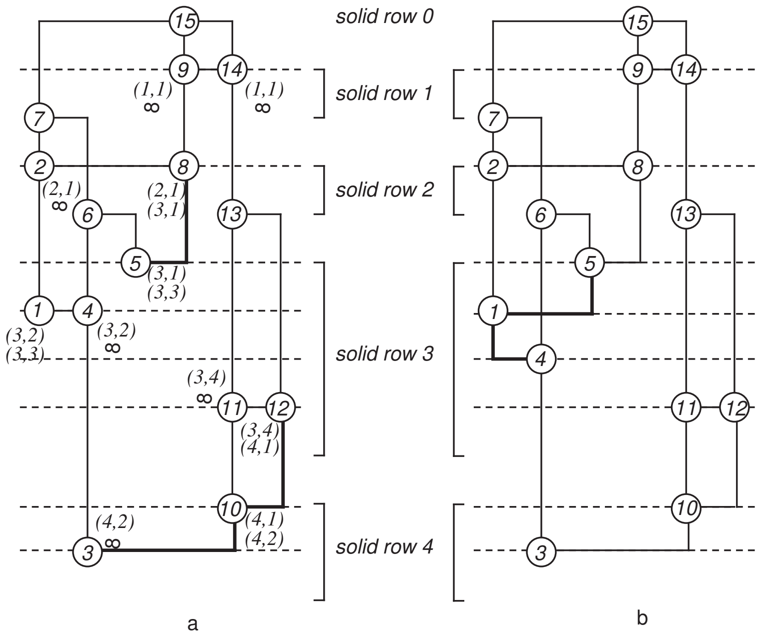

Figure 7: *A partial and the final drawing of G.*

5 Correctness and complexity

Theorem 5.1 *Given a cubic graph $G = (V, E)$, the algorithm described in the previous section correctly finds an orthogonal grid drawing for G in $O(\log n)$ time using n processors on the EREW PRAM model.*

Proof: To prove that the algorithm works correctly we only must prove that case c.4. covers all cases, since – if true – it is easy to see that the constructed drawing satisfies all the constrains of an orthogonal grid drawing.

First, observe that the sender always lies to the left of the receiver, by definition. Therefore, concerning the mutual position of sender and receiver, no other cases hold besides the two described in case c.4, that is either the sender lies below the receiver or it lies above the receiver. They cannot be in the same row because otherwise they would be connected, but the graph is simple, and double edges are not allowed. In all cases, up directions are never free and bottom directions are always free. So, the different cases depend on the freedom of the right and left directions. Also, notice that if the sender is higher than the receiver we always use the bottom position of the sender, while in the opposite case we always use the bottom position of the receiver. Therefore, the possible cases to be considered are the only ones depicted in Figs. 3 and 4, that is those described in Step 3.4.

For what concerns the complexity, it follows from the observations made in the previous section concerning the complexity of the individual steps of the algorithm. ∎

It is to notice that we need n processors only for the sorting phase.

Theorem 5.2 *Given a cubic graph $G = (V, E)$ with n vertices, the algorithm described in the previous section constructs an orthogonal grid drawing of G with at most $n + 1$ bends, $O(n)$ maximum edge length, and $(\frac{3}{4}n + 1) \times (\frac{3}{4}n + 1)$ maximum area. The number of bends on every edge is at most one.*

Proof: The orthogonal drawing is 1-bend by construction and then the length of every edge is $O(n)$.

Before proving the bounds for the total number of bends and the area, we define the value $f = t(r, v_1) + t(r, v_2) + t(r, v_3)$, that is the number of degree 3 vertices in $\mathcal{T}$ except for the root. f can also be seen as the number of "branches" in the tree.

Now, let us calculate a bound on the total number of bends. The first step of the algorithm consists in the embedding of $\mathcal{T}$ and produces exactly $f + 2$ bends, one for each "branch" plus two for embedding the children of the root. The second step represents on the grid all $\frac{n}{2} + 1$ edges in $\mathcal{L}$. Observe that each edge is drawn either as straight-line or with one bend. Only the cases in Figs. 3.b and 4.b introduces two bends globally, but both cases introduce one bend in the current edge and one bend in a non-tree edge that was previously drawn as straight-line. Consequently, drawing $\frac{n}{2} + 1$ non-tree edges introduces no more than $\frac{n}{2} + 1$ bends. All together, the whole orthogonal drawing of G contains at most $f + \frac{n}{2} + 3$ bends. This function is maximized at $f = \frac{n}{2} - 2$ and, therefore, the maximum number of bends is never more than $n + 1$.

Concerning the area of the drawing, we partition again the computation of the upper bound following the steps of the algorithm.

The width of the rectangle containing $\mathcal{T}$ is $f + 3$, since each "branch" introduces a new column and the root with its children introduces three columns. The height of the rectangle containing $\mathcal{T}$ is the height h of $\mathcal{T}$ itself, because each tree-level is represented on the same row. The height of $\mathcal{T}$ is no greater than $n - 2 - f$, and this value can be obtained starting from the worst case of a tree having the root with three children, whose two are leaves and one has degree two, and all the other vertices of $\mathcal{T}$ are connected in a path-like fashion to this vertex, that is all have degree two but the last one. The height of such a tree is $n-2$. Since each "branch" decreases the height of this tree at least by one, the inequality $h \leq n-2-f$ holds, with equality when all the vertices have degree either one or three.

The width is not increased by the second step. The height increases exactly by $\frac{n}{2}+1$, that is, by the sum of the thicknesses of all solid rows equal to the number of non-tree edges.

It follows that the area of the rectangle containing G is bounded by $(f+3)\times(\frac{3}{2}n-1-f)$. This function achieves its maximum at $f=\frac{3}{4}n-2$; in this case we have: $TotalArea \leq (\frac{3}{4}n+1)\times(\frac{3}{4}n+1)$. $\square$

6 Concluding remarks

Cubic graphs, or 3-regular graphs, are known to be useful tools for modeling a number of real-life situations [14, 5]. The main motivation for this work was to provide a simple and intuitive parallel algorithm that, given an arbitrary n-vertex cubic graph G as input, produces an orthogonal grid drawing of G. Such an algorithm could be of potential interest to VLSI design.

Consider the cubic graph G and let $\mathcal{T}$be a Breadth-First spanning tree of G. Our algorithm runs in $O(\log n)$ time, using n processors on a EREW PRAM. The resulting orthogonal grid drawing involves $O(n)$ bends, $O(n)$ maximum edge length and $O(n^2)$ area. Our algorithm matches the time and cost performance of the best previously-known algorithms while improving, at the same time, the constant factors involved in layout area and number of bends. However, more importantly, our algorithm stands out by its conceptual simplicity and ease of implementation.

References

[1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka, A simple parallel tree contraction algorithm, *Journal of Algorithms*, Vol. 10, 1989, pp. 287-302.

[2] A. Aho, J.K. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1973.

[3] R.J. Anderson and G.L. Miller, Deterministic parallel list ranking, *Algorithmica*, Vol. 6, 1991, pp. 859–868.

[4] T. Biedl and G. Kant, A better heuristic for orthogonal graph Drawings, *Computational Geometry: Theory and Applications*, Vol. 9, 1998 pp. 159–180. Also in *Proc. European Symposium on Algorithms (ESA '94)*, Lectures Notes in Computer Science 855, Springer-Verlag, 1994, pp 24–35.

[5] T. Calamoneri, *Does Cubicity Help to Solve Problems?*. Ph.D. Thesis, Univ. of Rome "La Sapienza", IX-2-97, 1997.

[6] T. Calamoneri, S. Jannelli and R. Petreschi, Experimental comparison of graph drawing algorithms for cubic graphs, *Journal of Graph Algorithms and Applications*, Vol. 3, No. 2, 1999, pp. 1–23.

[7] T. Calamoneri and R. Petreschi, An efficient orthogonal grid drawing algorithm for cubic graphs, *Proc. COCOON'95*, Lectures Notes in Computer Science 959, Springer-Verlag, 1995, pp. 31–40.

[8] T. Calamoneri and R. Petreschi, A parallel algorithm for orthogonal grid drawings of cubic graphs, *Journal of Parallel and Distributed Computing*, Vol. 55, 1998, pp. 94-108.

[9] R. Cole, Parallel merge sort, *SIAM Journal on Computing*, Vol. 17, No. 4, 1988, pp. 770–785.

[10] R. Cole and U. Vishkin, Approximate parallel scheduling. Part 1: the basic technique with applications to optimal parallel list ranking in logarithmic time," *SIAM Journal on Computing*, Vol. 18, 1988, pp. 128–142.

[11] G. Di Battista, P. Eades, R. Tamassia and J. Tollis, Algorithms for drawing graphs: an annotated bibliography, *Computational Geometry: Theory and Applications*, Vol. 4, No. 5, 1994, pp. 235–282.

[12] G. Di Battista, G. Liotta and F. Vargiu: Spirality of orthogonal representations and optimal drawings of series-parallel graphs and 3-planar graphs. *SIAM Journal on Computing*, Vol. 27, No. 6, 1998, pp. 1764–1811.

[13] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge Univ. Press, 1989.

[14] R. Greenlaw and R. Petreschi, Cubic graphs, *ACM Computing Surveys*, Vol. 27, No. 4, 1995, pp. 471–495.

[15] J. Jájá, *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.

[16] J. Jájá and J. Simon, Parallel algorithms in graph theory: planarity testing, *SIAM Journal on Computing*, Vol. 11, 1982, pp. 314–328.

[17] G. Kant, Drawing planar graphs using the canonical ordering, *Proc. 33th Annual IEEE Symposium on Foundations of Computer Science (FOCS'92)*, 1992, pp.101–110. Revised version in *Algorithmica - Special Issue on Graph Drawing* Vol. 16, 1996, pp. 4–32.

[18] Y. Liu, P. Marchioro and R. Petreschi, At most single bend embedding of cubic graphs, *Applied Mathematics*, Vol. 9, No. 2 ser.B, 1994, pp. 127–142.

[19] G.L. Miller and J.H. Reif, Parallel tree contraction and its applications, *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS'85)*, 1985, pp. 478–489.

[20] A. Papakostas and I.G. Tollis, Improved algorithms and bounds for orthogonal drawings, *Proc. Graph Drawing '94*, Lectures Notes in Computer Science 894, Springer-Verlag, 1994, pp. 40–51.

[21] M.S. Rahman, S. Nakano and T. Nishizeki, A linear-time algorithm for optimal orthogonal drawings of triconnected cubic plane graphs, *Proc. Graph Drawing '97*, Lectures Notes in Computer Science 1353, Springer-Verlag 1997, pp. 99–110.

[22] R. Tamassia, I.G. Tollis and J.S. Vitter, Lower bounds and parallel algorithms for planar orthogonal grid drawings, *Proc. IEEE Symp. Parallel and Distributed Processing*, 1991, pp. 1–8.

[23] R. Tamassia and J.S. Vitter, Parallel transitive closure and point location in planar structures, *SIAM J ournal on Computing*, Vol. 20, No. 4, 1991, pp. 708–725.

[24] J.C. Wyllie, *The Complexity of Parallel Computations*, Ph.D. Thesis, Cornell Univ., Dept. Computer Science, Ithaca, NY, 1979.