ELSEVIER

Information Processing Letters 67 (1998) 21–28

Information Processing Letters

A data-parallel algorithm for minimum-width tree layout ☆

Wuu Yang [1]

Department of Computer and Information Science, National Chiao-Tung University, Hsing Chu, Taiwan

Received 6 May 1997; received in revised form 17 April 1998
Communicated by D. Gries

Abstract

The tree-layout problem is to compute the coordinates of nodes of a tree so that the tree, when drawn on a piece of paper, appeals to human understanding. The tree-layout problem, which seems inherently sequential at the first glance, can be solved with a data-parallel algorithm. It takes O(*height* × log *width*) time on *width* processors when proper communication links between processors are available, where *height* and *width* are the height and width of the tree, respectively. The layout calculated by the algorithm has the minimum width. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Algorithms; Data-parallel algorithms; EREW; PRAM; Tree layout

1. Introduction

Traditional parallel algorithms are mostly applied in numerical computation, in which arrays are the most common data structure. Many new parallel algorithms have been discovered that are applicable to more flexible data structures. Furthermore, many problems that seem inherently sequential at the first glance are found to be solvable with parallel algorithms. We report in this paper that the tree-layout problem can also be solved with a data-parallel algorithm. The layout algorithm is applied to a tree structure, rather than the common rectangular arrays.

Parallel computers with tens of thousands of processors are typically programmed in a data-parallel fashion [3]. Data parallelism is similar to the SIMD (single instruction stream, multiple data stream) model of parallel machine architectures, in which the same operations are applied to multiple pieces of data simultaneously. It employs fine-grained and massive parallelism that exists in some kinds of applications. The applications include line drawing in computer graphics, VLSI design and circuit simulation, finite difference, multimedia image processing, and computer vision [4].

Trees are a common data structure widely used in computer science. For instance, the single-inheritance class hierarchy of an object-oriented programming system and directories of file systems are usually organized into tree structures. Sometimes, the tree structures are more comprehensible when they are drawn on a piece of paper or on the computer screen. The tree-layout problem is to compute the coordinates of nodes of a tree when the tree is drawn on a piece of paper. The layout is constrained to meet certain aesthetic standards [1] so that the tree, when drawn on a piece of paper, appeals to human understanding. Different aesthetic standards may be employed. In this paper, the standard is that (1) nodes on the same level (the *level* of a node will be defined in the

☆ This work was supported in part by National Science Council, Taiwan, R.O.C. under grant NSC 84-2213-E-009-043.

[1] Email: wuuyang@cis.nctu.edu.tw.

0020-0190/98/$19.00 © 1998 Elsevier Science B.V. All rights reserved.
PII: S0020-0190(98)00081-7

next section) should be drawn on the same horizontal line; (2) nodes on the same level should be separated from one another by a distance of at least one unit; and (3) the horizontal coordinate of a node must be equal to the mean of the horizontal coordinates of its leftmost and rightmost children. Applications of the layout algorithm include class browsers, graphic user interfaces of file managers, and software analysis [2].

The remainder of this paper is organized as follows: The next section presents the data-parallel layout algorithm. The third section outlines the proof that the layout algorithm satisfies the aesthetic standard. It is also shown that the layout calculated by the algorithm has the minimum width. Section 4 presents an improvement to the layout algorithm of Section 2. The improvement attempts to distribute additional horizontal spaces between nodes equally. The layout computed by the improved algorithm still satisfies the aesthetic standard; however, the layout may no longer have the minimum width. The last section concludes this paper and discusses related work.

2. A data-parallel tree-layout algorithm

We assume that a tree is represented by a linked-list data structure. Each node contains pointers to its parent and its leftmost and rightmost children. The *level* of a node is the level of its parent plus one; the root is on level one. (The notion of levels differs from the traditional notion of tree heights in that levels are concerned with nodes in the trees, whereas heights are concerned with edges or distance from the root. Since we want to calculate the coordinates of nodes, we adopt the notion of levels.) Nodes on the same level are called *cousins* of one another. Cousins are a generalized notion of siblings. Each node is assigned a processor, which is responsible for calculating the node's position. We also assume that there are sufficient processors and hardware links between the processors. The algorithm needs to determine the placement of the nodes.

The placement of nodes is determined by the vertical and horizontal coordinates. The tree is drawn upside down, that is, the root is at the top and descendants are drawn below ancestors. We ignore the sizes of the nodes in the algorithm; however, we will show a way to accommodate node sizes later. Since the distance between adjacent levels is a constant, the vertical coordinate of a node is simply the level of the node times the constant. Therefore, it suffices to compute the horizontal coordinates of nodes.

Before presenting the algorithm, we define a new operation:

$$\delta(a,b)=b \quad \text{if } b \neq \bot;$$

$$\delta(a,b)=a \quad \text{if } b = \bot,$$

where $\bot$ is a special value different from any integer. It can be verified that δ is an associative operation. That is,

$$\delta(\delta(a,b),c)=\delta(a,\delta(b,c)).$$

Then we define that

$$\mathop{\Delta}_{j=0}^{0} g_j = g_0$$

and

$$\mathop{\Delta}_{j=0}^{i} g_j = \delta\Big(\mathop{\Delta}_{j=0}^{i-1} g_j, g_i\Big) \quad \text{for } i > 1$$

(that is, $\Delta_{j=0}^{i} g_j = \delta(\delta(\cdots\delta(\delta(g_0,g_1),g_2),\ldots),g_i)$). $\Delta_{j=0}^{i} g_j$ is the rightmost non-$\bot$ value in the sequence $g_0, g_1, \ldots, g_i$, or is $\bot$ if all these g values are $\bot$.

The tree-layout algorithm consists of two phases, as shown in Fig. 1. Both phases proceed level by level: The first phase proceeds upwards from the bottom level to the root; the second downwards from the root to the bottom level. Consider a level s. Let $n_1, n_2, n_3, \ldots, n_\alpha$ be the nodes on level s, from left to right, where α is the number of nodes on level s. Each node n_i contains twelve variables c_i, d_i, e_i, f_i, g_i, h_i, k_i, l_i, m_i, p_i, q_i, and r_i. During the first phase, for each level s, each of the nine steps are performed by all nodes n_i on level s in parallel.

The c values of nodes are the tentative horizontal positions of the nodes *if* all their descendants are ignored. We adopt the convention that if node n_i does not have children, then $d_i = e_i = 0$. The f values of the nodes are their horizontal positions determined by their respective children. The g values are the numbers of units by which nodes are forced to shift to the right by their children. By convention, $g_0 = 0$. Note that if a node is forced to shift to the right by g units, its right cousins are also forced to shift to the right by at least g units. The h values result from propagating the g values to the right across nodes without children.

```
Algorithm: Tree-Layout
level := the height of tree
/* phase 1 */
for s := level down to 1 do
   Let n_1, n_2, n_3, ..., n_α be the nodes on level s (from left to right).
   step 1:  parallel for all nodes n_i on level s do  c_i := i od
   step 2:  parallel for all nodes n_i on level s do  d_i := the m value of the leftmost child of node n_i od
   step 3:  parallel for all nodes n_i on level s do  e_i := the m value of the rightmost child of node n_i od
   step 4:  parallel for all nodes n_i on level s do  f_i := (d_i + e_i) / 2 od
   step 5:  parallel for all nodes n_i on level s do  g_i := if n_i has children then f_i − c_i else ⊥ od
   step 6:  parallel for all nodes n_i on level s do  h_i := Δ_{j=0}^{i} g_j od (Assume g_0 = 0.)
   step 7:  parallel for all nodes n_i on level s do  k_i := maximum(h_{i−1} − h_i, 0) od (Assume h_0 = 0.)
   step 8:  parallel for all nodes n_i on level s do  l_i := Σ_{j=1}^{i} k_j od
   step 9:  parallel for all nodes n_i on level s do  m_i := c_i + h_i + l_i od
od
/* phase 2 */
the q value of the root := 0
the r value of the root := the m value of the root
for s := 2 to level do
   Let n_1, n_2, n_3, ..., n_α be the nodes on level s (from left to right).
   step 10: parallel for all nodes n_i on level s do  p_i := the q value of node n_i's parent od
   step 11: parallel for all nodes n_i on level s do  q_i := l_i + p_i od
   step 12: parallel for all nodes n_i on level s do  r_i := m_i + p_i od
od
```

Fig. 1. The data-parallel tree-layout algorithm.

By convention, $h_0 = 0$. The k value of a node n_i is the number of units by which n_i's descendants are forced to shift to the right by the descendants of n_i's nearest left cousin that has children. The l value of a node n_i is the *cumulative* amount of right-shift for n_i's descendants forced by the descendants of n_i's left cousins. The m value of a node n_i is the horizontal coordinate of n_i if all nodes above the current level are ignored.

Suppose that node N is on level s and that $t_1, t_2, \ldots, t_{s-1}$ are N's ancestors. N's horizontal coordinate is

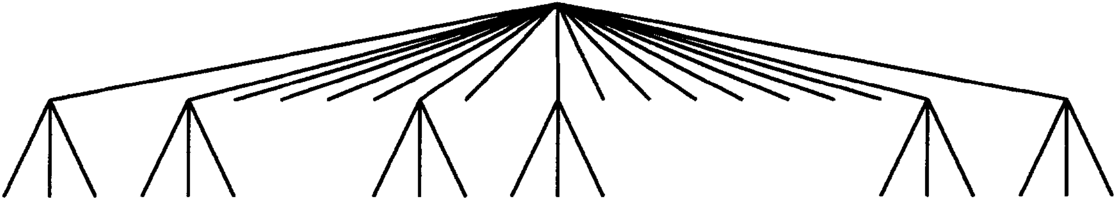
$$m_n + \sum_{j=1}^{s-1} l_j,$$

where m_n is the m value of node N and l_j is the l value of node t_j. The second phase calculates the horizontal coordinates of all nodes level by level. The p and q values of the nodes are used to compute $\sum_{j=1}^{s-1} l_j$. The r values are the final horizontal coordinates of the nodes.

Example. Consider the tree in Fig. 2(a). There are three levels in the tree. The first phase of the layout algorithm is applied to the tree from level 3 to level 1. It is easy to verify that the m values of nodes on the third level are $1, 2, \ldots, 18$. Consider the nodes on the second level. For convenience, the nodes on the second level are named $n_1, n_2, \ldots$, and n_{18}. The c, f, g, h, k, l, m, p, q, and r values for these nodes are listed in the table in Fig. 2(b). The m value of the root is 13. Therefore, the horizontal coordinate of the root is 13; the horizontal coordinate of the rightmost node on the second level is 24; the horizontal coordinate of the rightmost node on the third level is 25. The tree that is drawn by the layout algorithm is shown in Fig. 2(a).

Note that $g_2 = 3$, which means that node n_2 is forced to shift to the right by three units by its descendants. Hence, all the right cousins of n_2 are

(a) An example tree.



(b) Computing the horizontal locations of nodes on the second level.

nodes n_i	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}	n_{13}	n_{14}	n_{15}	n_{16}	n_{17}	n_{18}
c_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
f_i	2	5	0	0	0	0	8	0	11	0	0	0	0	0	0	0	14	17
g_i	1	3	$\perp$	$\perp$	$\perp$	$\perp$	1	$\perp$	2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	-3	-1
h_i	1	3	3	3	3	3	1	1	2	2	2	2	2	2	2	2	-3	-1
k_i	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	5	0
l_i	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	7	7
m_i	2	5	6	7	8	9	10	11	13	14	15	16	17	18	19	20	21	24
p_i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q_i	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	7	7
r_i	2	5	6	7	8	9	10	11	13	14	15	16	17	18	19	20	21	24

Fig. 2. An example demonstrating the data-parallel layout algorithm.

forced to shift to the right by at least three units. Since $g_7 = 1 < g_2$ (hence $k_7 = 2$), n_7's descendants are forced to shift to the right by two units by n_2's descendants. Consequently, the descendants of n_7's right cousins are all forced to shift to the right by at least two units. (That is, $l_i \geqslant 2$ for $i \geqslant 7$.) Similar arguments apply to node n_{17}.

To analyze the time complexity, we assume a PRAM EREW model [6]. Consider the time spent on each level first. As explained in [3], the parallel prefix-sum operation at step 8 takes $O(\log w)$ time, where w is the number of nodes on that level. The parallel prefix-Δ operation at step 6, which finds the g value of the nearest non-$\perp$ left cousin, is similar to the parallel prefix-sum operation. It also takes $O(\log w)$ time. To determine whether a node is the leftmost child of its parent, we assume that every node knows the identity of its parent. Each node compares the identity of the parent of its left cousin against the identity of its own parent. If the two differ, the node is the leftmost child of its parent. This operation takes constant time. Similar arguments apply to the rightmost children. We assume that a node can broadcast an integer to all its children in one time unit at step 10. All other operations take constant time. Therefore, $O(\log w)$ time is spent on a level. The total time needed is $O(\mathit{height} \times \log \mathit{width})$, where *height* is the number of levels and *width* is the maximum number of nodes on any level. Since the algorithm works level by level, the processors may be shared by nodes on different levels. Therefore, the total number of processors needed is *width*.

Up to now, we have assumed that nodes are represented by points on a piece of paper. In practice, nodes may be represented by rectangles. It is easy to incorporate the widths of nodes. Node widths may be added to c_i's and proper adjustment may be made before calculating d_i's. To accommodate the heights of nodes, an easy method is to make all nodes on a level as tall as the tallest nodes on that level. The vertical distance between adjacent levels is adjusted accordingly. This method may not be satisfactory in cases where node heights vary dramatically; however, it does align nodes on the same level on the same horizontal line.

3. Correctness and the minimum-width property

In this section, we show that the layout algorithm satisfies the aesthetic standard proposed in the first section and that the layout calculated by the algorithm has the minimum width. It is obvious that nodes on the same level are drawn on the same horizontal line (the first criterion). We need to show that nodes on the same level are separated from one another by a distance of at least one unit and that the horizontal coordinate of a node is equal to the mean of the horizontal coordinates of its leftmost and rightmost children. For the sake of brevity, the detailed proofs are omitted.

Lemma 1. *On any level,* $0 \leqslant l_i \leqslant l_j$ *for* $i \leqslant j$.

Lemma 2. *On any level,* $h_i + l_i \leqslant h_j + l_j$ *for* $i \leqslant j$.

Theorem 3. *On any level,* $r_i \leqslant r_{i+1} + 1$ *for all* i.

Theorem 4. *The horizontal coordinate of a node is equal to the mean of the horizontal coordinates of its leftmost and rightmost children.*

Finally, we wish to show that the layout computed by the data-parallel algorithm is of high quality. Specifically, we show that the computed layout has the narrowest width among all layouts that satisfy the aesthetic standard. We need a new terminology in proving the assertion. A *forest* is an (ordered) sequence of trees. We may imagine that the trees in the forest are the subtrees of a fictitious root. The fictitious root together with the trees in the forest may be laid out as any other trees. A *forest layout* is a layout of the sequence of the trees in the forest.

Definition. The *width* of a forest layout is the horizontal distance between the leftmost and the rightmost nodes in the layout.

Note that the width of a layout of a forest is *not* necessarily the sum of the widths of the layouts of individual trees. A layout of a tree automatically induces a layout for every subtree of the tree. The following lemma is a direct consequence of the aesthetic standard.

Lemma 5. *If* L *is a layout for a tree* T *that satisfies the aesthetic standard, then the layout for any subtree of* T *induced by* L *also satisfies the aesthetic standard.*

In what follows, the word *layout* always implicitly implies that the layout satisfies the aesthetic standard mentioned in the first section. For a tree, there might be more than one minimum-width layout. Therefore, we choose one among them as a *thin* layout.

Definition. A layout L for a tree is a *thin* layout if it has the minimum width and the layout for any subtree of the tree induced by L also has the minimum width.

Note that there might be more than one thin layout for a tree. For instance, in Fig. 2(a), the eighth node (from left) on the second level can be moved to the right one unit and is still a thin layout.

Theorem 6. *The layout for any tree computed by the algorithm is a thin layout.*

Proof (*Sketch*). Consider the algorithm in Fig. 1. At the end of each iteration during the first phase, we obtain a forest and a layout for the forest. Let F_i and L_i denote the forest and the forest layout obtained at the end of the ith iteration, respectively. Without a detailed proof, we claim that the width of the forest layout obtained in one iteration must be no less than the width of the forest layout obtained during the previous iteration. □

Specifically, we proved the following assertion by an inductive argument:

Assertion. *For all* i, *at the end of the* i*th iteration,*
(1) *the layout* L_i *has the minimum width among all the possible layouts for the forest* F_i; *and*
(2) *the layout for every tree in* F_i *induced by* L_i *is a thin layout.*

Corollary. *For any tree, the layout computed by the algorithm has the minimum width.*

4. An improvement

Note that in Fig. 2(a), the eighth node (from left to right) on the second level, that is, node n_8 in Fig. 2(b), is placed one unit to the right of node n_7 and two units to the left of node n_9. Though this arrangement satisfies our aesthetic standard, it would be preferable to place node n_8 in the middle of n_7 and n_9. In this section, we present an improvement of the layout algorithm of Section 2. In the improved algorithm, shown in Fig. 3(a), nodes are placed equally spaced if the arrangement does not violate the aesthetic standard. Five steps are added to the first phase in the improved algorithm. We first define two operations λ and θ.

Define $\lambda(a,b) =$ *if* $b \leqslant 0$ *then* b *else if* $a \leqslant 0$ *then* $a - b$ *else* $a + b$. It can be verified that λ is an associative operation. In step 10 in Fig. 3(a), we assume $u_0 = 0$ and define

$$\mathop{\Lambda}_{j=0}^{0} u_j = u_0$$

and

$$\mathop{\Lambda}_{j=0}^{i} u_j = \lambda\Big(\mathop{\Lambda}_{j=0}^{i-1} u_j, u_i\Big) \quad \text{for } i \geqslant 1.$$

Define $\theta(a,b) =$ *if* $a \leqslant 0$ *then* a *else if* $b = 0$ *then* $-a$ *else* b. It can be verified that θ is also an associative operation. Let α be the number of nodes in the current level. In step 12, we assume $w_{\alpha+1} = 0$ and define

$$\mathop{\Theta}_{j=\alpha+1}^{\alpha+1} w_j = w_{\alpha+1}$$

and

$$\mathop{\Theta}_{j=i}^{\alpha+1} w_j = \theta\Big(w_i, \mathop{\Theta}_{j=i+1}^{\alpha+1} w_j\Big) \quad \text{for } 1 \leqslant i \leqslant \alpha.$$

In step 9, the sequence $u_1, u_2, \ldots$ is a sequence of 0's and 1's depending on whether the nodes $n_1, n_2, \ldots$ have children. This sequence is divided into one or more segments by the 0's in the sequence. Each segment contains zero or more 1's and is delimited by 0's on both ends. In step 10, the operation $\Lambda_{j=0}^{i} u_j$ computes the negation of the position of node n_i in its segment. For instance, given the sequence of u_i's $1, 1, 0, 1, 1, 1, 0, 1, 1$, the corresponding v_i's computed in step 10 are $-1, -2, 0, -1, -2, -3, 0, -1, -2$. The use of negation is for separating independent segments. Step 11 calculates the amount of right-shift for the nodes that have no children. Note that w_i computed in step 11 is actually one greater than the actual amount of right-shift. It can be proved that $w_i \geqslant 0$, for all i. Step 12 propagates the amount of right-shift from right to left within the same segment. Note that step 12 performs a parallel suffix-θ operation. The resulting x_i is either 0 or is a negative number. The addition of 1 to w_i (in step 11) and the negation used in x_i (in step 12) are for separating independent segments. Step 13 readjusts the amount of right-shift for each node. Finally, step 14 adjusts m_i's with y_i's.

Example. Fig. 3(b) shows the computation performed by the improved algorithm. Note that r_8 is 11.5, which means that n_8 is placed in the middle of n_7 and n_9.

5. Related works and conclusion

We have presented a data-parallel tree-layout algorithm. The algorithm meets the aesthetic standard. The layout calculated by the algorithm has the minimum width. The algorithm takes

$$O(\textit{height} \times \log \textit{width})$$

time on *width* processors when proper communication links between processors are available, where *height* and *width* are the height and width of the tree, respectively. An improvement to the layout algorithm is also presented, which attempts to distribute additional horizontal spaces among nodes on the same levels.

There are many published *sequential* tree-layout algorithms. The algorithm in [5] employs "wraps" around subtrees and attempts to combine these subtree wraps as close, but not overlapping, to each other as possible. The algorithm can handle the case that nodes may have different sizes. The algorithm in [9] employs a different aesthetic standard. It also draws a layout with the minimum width under the adopted aesthetic standard. Reingold and Tilford use a threaded technique to represent the boundary of a subtree [7]. The algorithm in [1] can handle the case that nodes may have different sizes. The algorithm in [8] consists of three phases that calculate the horizontal and vertical positions, respectively. All of the above mentioned algorithms are sequential. By contrast, this paper shows that the layout problem can be solved with a data-parallel algorithm.

(a) the improved layout algorithm

Algorithm: Improved-Tree-Layout

level := the height of *tree*

/* phase 1 */

for s := *level* **down to** 1 **do**

  Let $n_1, n_2, n_3, \ldots, n_\alpha$ be the nodes on level s (from left to right).

  step 1: **parallel for** all nodes n_i on level s **do** $c_i := i$ **od**

  step 2: **parallel for** all nodes n_i on level s **do** d_i := the m value of the leftmost child of node n_i **od**

  step 3: **parallel for** all nodes n_i on level s **do** e_i := the m value of the rightmost child of node n_i **od**

  step 4: **parallel for** all nodes n_i on level s **do** $f_i := (d_i + e_i)/2$ **od**

  step 5: **parallel for** all nodes n_i on level s **do** $g_i :=$ *if* n_i has children *then* $f_i - c_i$ *else* $\perp$ **od**

  step 6: **parallel for** all nodes n_i on level s **do** $h_i := \overset{i}{\underset{j=0}{\Delta}} g_j$ **od** (Assume $g_0 = 0$.)

  step 7: **parallel for** all nodes n_i on level s **do** $k_i :=$ *maximum* $(h_{i-1} - h_i, 0)$ **od** (Assume $h_0 = 0$.)

  step 8: **parallel for** all nodes n_i on level s **do** $l_i := \sum_{j=1}^{i} k_j$ **od**

  step 9: **parallel for** all nodes n_i on level s **do** $u_i :=$ *if* $g_i = \perp$ *then* 1 *else* 0 **od**

  step 10: **parallel for** all nodes n_i on level s **do** $v_i := \overset{i}{\underset{j=0}{\Lambda}} u_j$ **od** (Assume $u_0 = 0$.)

  step 11: **parallel for** all nodes n_i on level s **do** $w_i :=$ *if* $g_i = \perp$ and $h_i < h_{i+1}$ *then* $1 + (h_i - h_{i+1})/(v_i - 1)$ *else if* $g_i = \perp$ *then* 1 *else* 0 **od**

  step 12: **parallel for** all nodes n_i on level s **do** $x_i := \overset{\alpha+1}{\underset{j=i}{\Theta}} w_j$ **od** (Assume $w_{\alpha+1} = 0$.)

  step 13: **parallel for** all nodes n_i on level s **do** $y_i :=$ *if* $x_i < 0$ and $-x_i - 1$ *else* 0 **od**

  step 14: **parallel for** all nodes n_i on level s **do** $m_i := c_i + h_i + l_i + y_i$ **od**

od

/* phase 2 */

the q value of the root := 0

the r value of the root := the m value of the root

for s := 2 **to** *level* **do**

  Let $n_1, n_2, n_3, \ldots, n_\alpha$ be the nodes on level s (from left to right).

  step 15: **parallel for** all nodes n_i on level s **do** p_i := the q value of node n_i's parent **od**

  step 16: **parallel for** all nodes n_i on level s **do** $q_i := l_i + p_i$ **od**

  step 17: **parallel for** all nodes n_i on level s **do** $r_i := m_i + p_i$ **od**

od

(b) Computing the horizontal locations of nodes on the second level by the improved algorithm.

nodes n_i	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}	n_{13}	n_{14}	n_{15}	n_{16}	n_{17}	n_{18}
c_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
h_i	1	3	3	3	3	3	1	1	2	2	2	2	2	2	2	2	-3	-1
l_i	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	7	7
u_i	0	0	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0
v_i	0	0	-1	-2	-3	-4	0	-1	0	-1	-2	-3	-4	-5	-6	-7	0	0
w_i	0	0	1	1	1	1	0	1.5	0	1	1	1	1	1	1	1	0	0
x_i	0	0	-1	-1	-1	-1	0	-1.5	0	-1	-1	-1	-1	-1	-1	-1	0	0
y_i	0	0	0	0	0	0	0	0.5	0	0	0	0	0	0	0	0	0	0
m_i	2	5	6	7	8	9	10	11.5	13	14	15	16	17	18	19	20	21	24
p_i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q_i	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	7	7
r_i	2	5	6	7	8	9	10	11.5	13	14	15	16	17	18	19	20	21	24

Fig. 3. The improved data-parallel tree-layout algorithm.

The data-parallel tree-layout algorithm presented in this paper is motivated by the sequential tree-layout algorithm in [9]. The result obtained from the parallel algorithm is similar, but not identical, to that obtained by Wetherell and Shannon's algorithm. It would be interesting to study the mechanical transformation of sequential algorithms into (data-)parallel algorithms.

References

[1] A. Bloesch, Aesthetic layout of generalized trees, Software—Practice and Experience 23 (8) (1993) 817–827.

[2] E.R. Gansner, S.C. North, K.P. Vo, Graph visualization in software analysis, in: Proc. 1992 Symposium on Assessment of Quality Software Development Tools, 1992.

[3] W.D. Hillis, G.L. Steele Jr, Data parallel algorithms, Comm. ACM 29 (12) (1986) 1170–1183.

[4] T.G. Lewis, H. El-Rewini, Introduction to Parallel Computing, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[5] S. Moen, Drawing dynamic trees, IEEE Software 7 (4) (1990) 21–28.

[6] M.J. Quinn, Parallel Computing: Theory and Practice, McGraw-Hill, New York, 1994.

[7] E.M. Reingold, J.S. Tilford, Tidier drawings of trees, IEEE Trans. Software Engineering 5 (5) (1981) 514–520.

[8] J.G. Vaucher, Pretty-printing of trees, Software—Practice and Experience 10 (1980) 553–561.

[9] C. Wetherell, A. Shannon, Tidy drawings of trees, IEEE Trans. Software Engineering 5 (5) (1979) 514–520.