Computer and Information Science; Vol. 13, No. 1; 2020
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Sort Integers into a Linked List

Yijie Han[1], Hemasree Koganti[1], Nikita Goyal[1]

[1] School of Computing and Engineering, University of Missouri at Kansas City, United States

Correspondence: Yijie Han, School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, MO 64110, United States.

Received: October 17, 2019 Accepted: January 6, 2020 Online Published: January 14, 2020

doi:10.5539/cis.v13n1p51 URL: https://doi.org/10.5539/cis.v13n1p51

Abstract

We show that n integers in {0, 1, …, m-1} can be sorted into a linked list in constant time using nlogm processors on the Priority CRCW PRAM model, and they can be sorted into a linked list in O(loglogm/logt) time using nt processors on the Priority CRCW PRAM model.

Keywords: sorting, linked list, integer sorting, parallel algorithms

1. Introduction

It is well known that Ω(logn/loglogn) is a time lower bound for sorting integers (P. Beame & J. Histad,1989). However, if we sort integers into a linked list this lower bound needs not hold. Sorting integers into a linked list is to let smaller integers precede larger integers in the linked list. It is known that n integers in {0, 1, …, m-1} can be sorted into a linked list in O(loglogm) time using n processors on the CRCW PRAM (P.C.P. Bhatt, K. Diks, T. Hagerup, T. Radzik, S. Saxena, 1991). As in approximate sorting (T. Goldberg & U. Zwick, 1995) (T. Hagerup & R. Raman,1993) we may allow padding when sort integers into a linked list. It is known that n 0-1's can be sorted into a linked list by chaining 0's into a linked list and 1's into another linked list. This can be done in α(n) time using n/α(n) processors (P. Ragde,1993), where α(n) is the inverse Ackermann function. Sort padded 0-1 into a linked list takes constant time with n processors. This can be done by making a dummy 0 for each 1 and a dummy 1 for each 0 and then chaining 0's and dummy 0's into a linked list and 1's and dummy 1's into another linked list. Sort integers into a linked list has resulted faster and efficient parallel algorithms for sorting integers in an array (Y. Han & X. Shen, 2002).

In this paper we show that n integers in {0, 1, …, m-1} can be sorted into a linked list in constant time using nlogm processors on the Priority CRCW PRAM model, and they can be sorted into a linked list in O(loglogm/logt) time using nt processors on the Priority CRCW PRAM model.

The computation model used in this paper is the CRCW (Concurrent Read Concurrent Write) PRAM (Parallel Random-Access Machine) Model (R. M. Karp & V. Ramachandran, 1991). On the CRCW PRAM memory is shared among processors and multiple processors can read the same memory cell in one step and can write to the same memory cell in one step. When concurrent write happens, we use the Priority CRCW PRAM (T. Hagerup,1987) in which when multiple processors write the same cell in one step the highest indexed processor wins the write.

We use a trie of height logm to sort integers into a linked list. We use A[i][j] to represent the j-th node of the trie at level i. When i=0, A[0][j] is a node at a leaf of the trie. When i>0 then A[i-1][2j] is the left child of A[i][j] and A[i-1][2j+1] is the right child of A[i][j]. 0 is labeled on the edge from a parent to its left child and 1 is labeled on the edge from a parent to its right child. The label reads from the root of the trie to leaf A[0][j] is the binary representation of j. Without loss of generality we assume that logm is a power of 2 as when this is not true, we use the smallest power of 2 greater than logm in place of logm. A trie of 16 leaves is shown in Fig. 1.

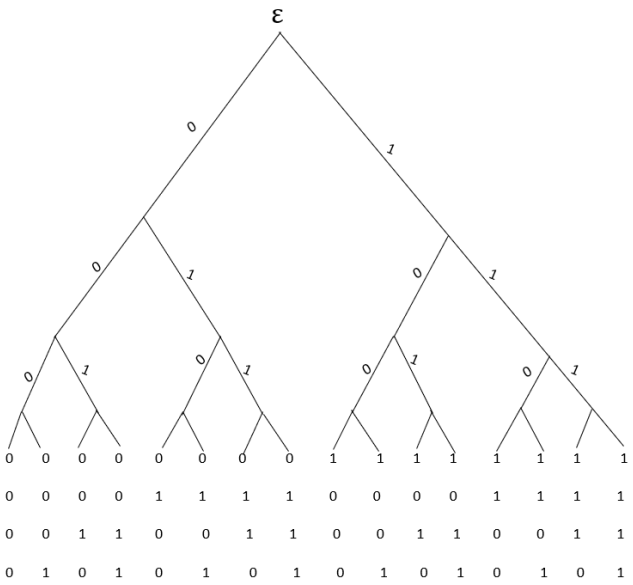

Figure 1. Example of a Trie

When nt processors are used the trie with height logm is divided into t section's and each number a at the leaf of the trie will use 1 of the t processors (concurrently with processors for other leaves of the trie) to write into the A[ilogm/t][a div $2^{ilogm/t}$], i=1, …, t, where div is the integer division.

2. The Algorithm with nlogm Processors

Let I be the input array of n integers in {0, 1, …, m-1}. I[i] is first placed in A[0][I[i]] at the leaf of the trie. We assume that all input integers are distinct for otherwise we will replace I[i] with I[i]*n+i as the input integer. The input integers and processors assigned to them are shown in Fig. 2.

Integer	0	5	3	9	4	8
Processor	0	1	2	3	4	5

Figure 2. Input array with processors assigned

We will build a tree for the input integers based on the trie. An interior node of the tree is a node in trie such that the node has a left child and a right child. Node having a single child in the tree is removed. Such a tree is shown in Fig 3. The reason such a tree is built is because the tree can facilitate searching and finding the predecessor and successor of an integer (Y. Han & H. Koganti, 2018).

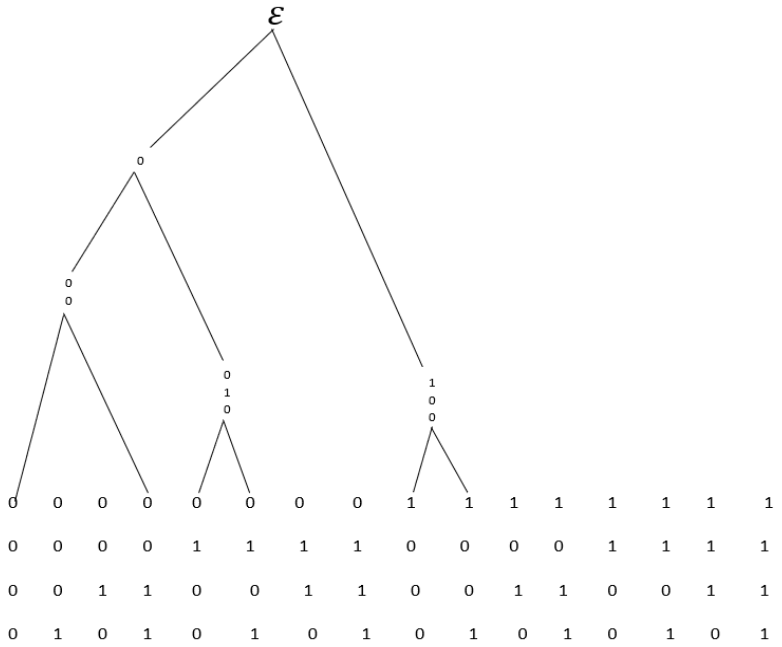

Figure 3. The nodes with one child are removed.

This tree is constructed in constant time with nlogm processors on the Priority CRCW PRAM and in O(loglogn/logt) time with nt processors on the Priority CRCW PRAM.

When we use nlogm processors we allocate logm processors for each input integer. I[i] will use the j-th processor, $1 \leq j \leq \log m$, at A[j][I[i] div 2^j], where a div b = $\lfloor a/b \rfloor$. Processors at A[i][j] will first use concurrent write to write it's processor id (index) into A[i][j]. Then the processor(s) at A[i][j] will check if A[i-1][2j] and A[i-1][2j+1] are written. This determines whether A[i][j] has one child or has two children. We will label A[i][j] with 1 if it has two children and label A[i][j] with 0 if it has one child. The leaves are always labeled with 1. This situation is depicted in Fig. 4.

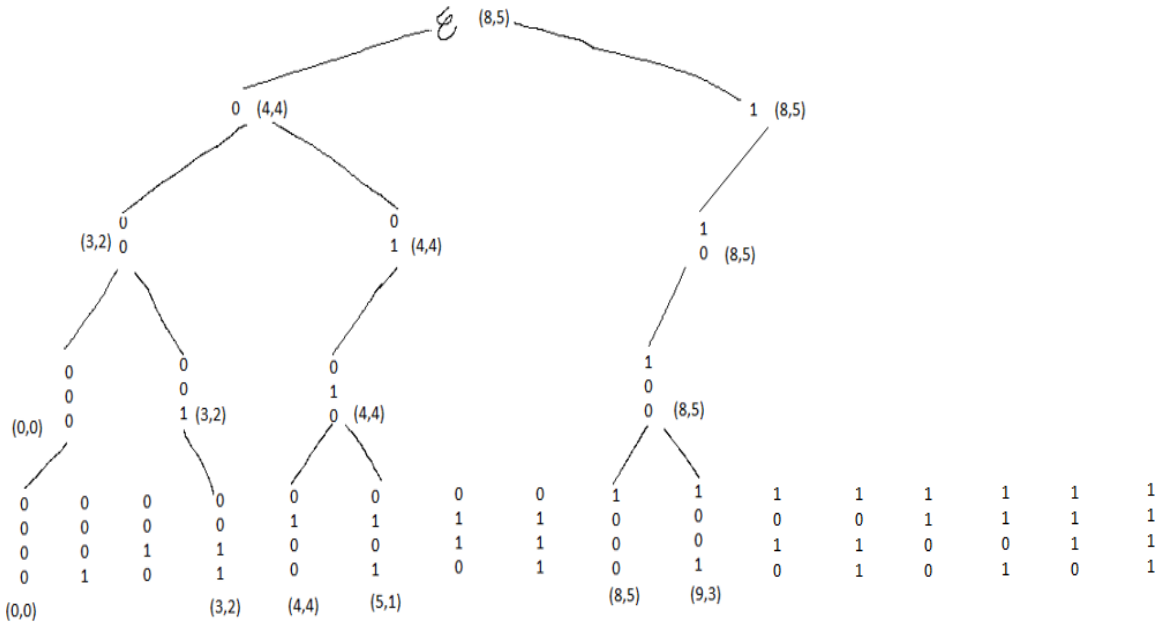

Figure 4. All the nodes are shown

Now for the root of A and each node in A that is labeled with 1 we need to find its nearest descendants that are labeled with 1. Say A[i][j] is labeled with 1 and processor p wins the concurrent write at A[i][j]. Let A[i'][j'] and A[i''][j''] are the two nearest descendants of A[i][j] that are labeled with 1's. If an integer a is a leaf of A[i'][j'] (A[i''][j'']) then because we use Priority CRCW PRAM the processors associated with a win the write at A[i'][j'] (A[i''][j'']) and all the ancestors of A[i'][j'] (A[i''][j'']) up to A[i][j]. And another integer b at the leaf of A[i''][j''] (A[I'][j']) and the processors associated with b win the write at A[i''][j''] (A[I'][j']) and all ancestors of A[i''][j''] (A[I'][j']) up to A[i][j]. If the processor associated with a (b) wins the concurrent write at A[i][j] then we will use the logm processors associated with b (a) to link A[i'][j'] and A[i''][j''] to A[i][j].

To link A[i'][j'] to A[i][j], these logm processors will form an array of size B[0..logm]. B[c], i≤c≤logm, will be set to -1. B[c], 0≤c<I, will be set to c if A[c][j'''] is labeled with 1, where A[c][j'''] is an ancestor of A[i'][j'] in the trie, it will be set to -1 if A[c][j'''] is labeled with 0. Then we use logm processors to find the maximum in array B. This takes constant time with logm processors (F.E. Fich, P. L. Ragde & A. Wigderson, 1988). The way to link A[i''][j''] to A[i][j] is similar. Thus, in constant time we build the tree for the input integers. The tree built is shown in Fig. 3.

To chain the integers into a linked list, we need to let each leaf a in the tree find the lowest ancestor in the tree that has a left (right) child which is not an ancestor of a. For leaf a to find the lowest ancestor in the tree that has a left child which is not an ancestor of a, we will use the logm processors for a, if a's ancestor at level l in trie is not a node in the tree (i.e. it has one child) then processor l will write logm+1 into array B[l]. Processor l will write logm+1 into B[l] also if the ancestor a' of a at level l of the trie has its left child which is an ancestor of a. Otherwise the ancestor a' of a at level l of the trie has its left child which is not an ancestor of a and processor l will write l into B[l]. Then we need to find minimum in array B which takes constant time with logm processors (F.E. Fich, P. L. Ragde, and A. Wigderson, 1988). This situation is shown in Fig. 3.

If leaf a locates b as the lowest ancestor in the tree that has a right child which is not an ancestor of a and a' locates b as the lowest ancestor in the tree that has a left child which is not an ancestor of a'' then we link a to a'. This builds the linked list for the input integers in constant time.

Theorem 1: n integers in {0, 1, …, m-1} can be sorted into a linked list in constant time with nlogm processors on the Priority CRCW PRAM.

As we noted (Y. Han & H. Koganti, 2018) that the tree built here can be augmented to facilitate predecessor and successor queries and insertion in O (loglogm) time.

3. The Algorithm with nt Processors

When we have nt<nlogm processors, we will assign t processors to each input integer. Integer a will be dropped at A[0][a] and the i-th processor for a will write at A[ilogm/t][a div $2^{ilogm/t}$]. Then processors for a will find the highest level in the trie that they win the write. Let this level be level l. Then all the t processors allocated to a will move to A[llogm/t][a div $2^{llogm/t}$]. This cuts the trie into t sections as shown in Fig. 6. Now the linked list in each subtrie is built recursively.

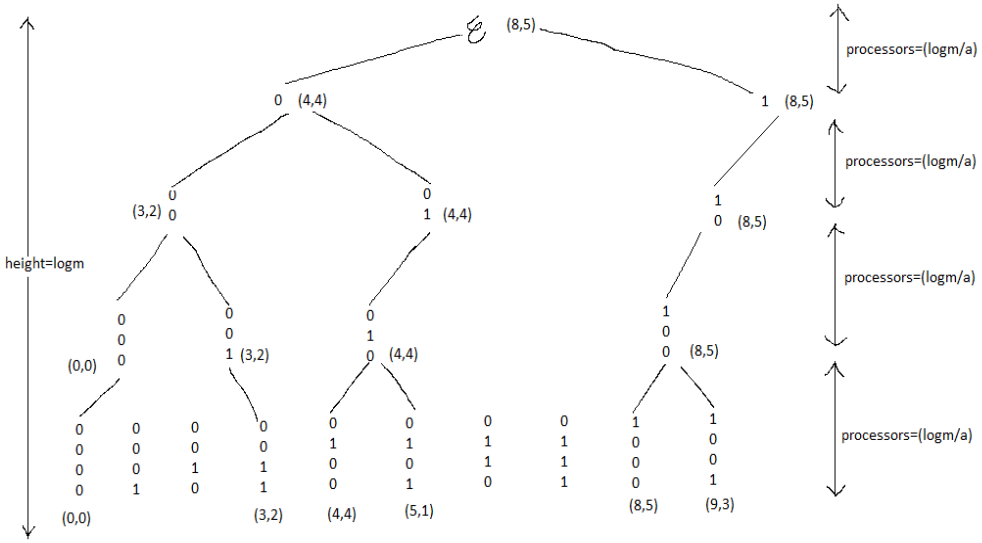

Figure 6. Trie divided into sections

After we return from the recursion the linked list for each subtrie is built. We said that processors for integer a was winning at A[llogm/t][a div $2^{llogm/t}$] and now the linked list for the subtrie (with logm/t levels) rooted at A[(l-1)logm/t][a div $2^{(l-1)logm/t}$] is built. Now a uses the b-th

processor and processors for the linked list at the subtrie rooted at A[blogm/t][a div $2^{blogm/t}$] to insert it into the linked list at the subtrie rooted at A[blogm/t][a div $2^{blogm/t}$]. Note that if the subtries rooted at A[blogm/t][a div $2^{blogm/t}$] and A[(b+1)logm/t][a div $2^{(b+1)logm/t}$] are empty then a will not insert into the empty linked list for the subtrie rooted at A[blogm/t][a div $2^{blogm/t}$].

Then the linked lists at t different levels will be joined into one linked list. This is done by letting (the largest (smallest) integer in) each linked list in the subtrie rooted at r' find the lowest ancestor having a leftmost right (rightmost left) child which is not an ancestor of r'. Then chaining the linked list as in the previous section.

Because there are O(logm/logt) levels of recursion and thus we build the linked list for the input integer in O(logm/logt) time.

Theorem 2: n integers in {0, 1, …, m-1} can be sorted into a linked list in O(logm/logt) time with nt processors on the Priority CRCW PRAM.

We can then build the tree for the linked list. This is done by assuming that when the recursion returns both linked list and the tree for the subtries are built. Because the linked list is built we can then insert the (processor associated with the) integer winning the write to the root of the subtrie back into the linked list in constant time by comparing all integers in the linked list with this winning integer. After inserting into the linked list we can compare this winning integer with its two neighboring integers to determining the lowest ancestor of this winning integer with its two neighbors. Thus this winning integer can be inserted into the tree in constant time. Then (the largest (smallest) integer in) in the tree of the subtrie rooted at r' find the lowest ancestors a' (a'') that has a leftmost right child c' (rightmost left child c'') which is not an ancestor of r'. The root of the tree in the subtrie rooted at r' will link to either a' or a'', whichever is at lower levels of the trie. This builds the tree for the trie.

4. Building a Tree Based on a Trie: An Example

We will build a binary tree for the n input integers based on a trie. For suppose we have 6 input integers 0,5,3,9,4 and 8 we assign a processor to each integer as shown in Fig. 2. and the processor will drop the integer at it's position as in Fig 4. Priority CRCW approach is used and the highest indexed processor will win the write and move upward. The same approach is used at every level and the highest index processor with its respective integer will reach the root like shown in Fig 2. The height of the trie will be logm. The intermediate nodes are removed and only the highest-level node will be placed in the tree as in Fig. 7.

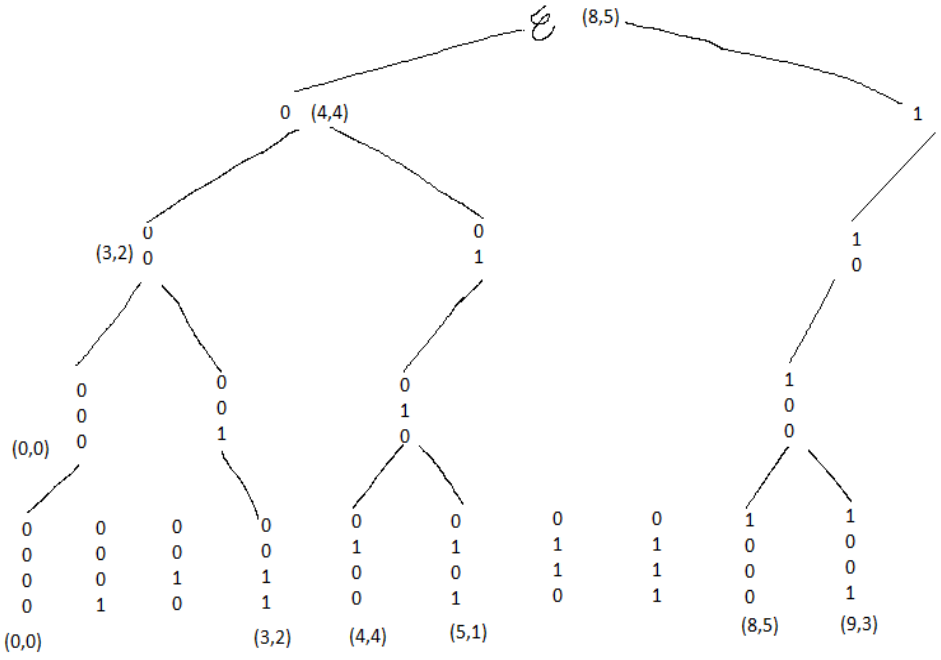

Figure 7. Intermediate levels are removed

For example (4,4) at nodes ‘010’ and ‘00’ can be removed and kept only at node ‘0’. At each level a linked list is created if there are any nodes like Fig 5.

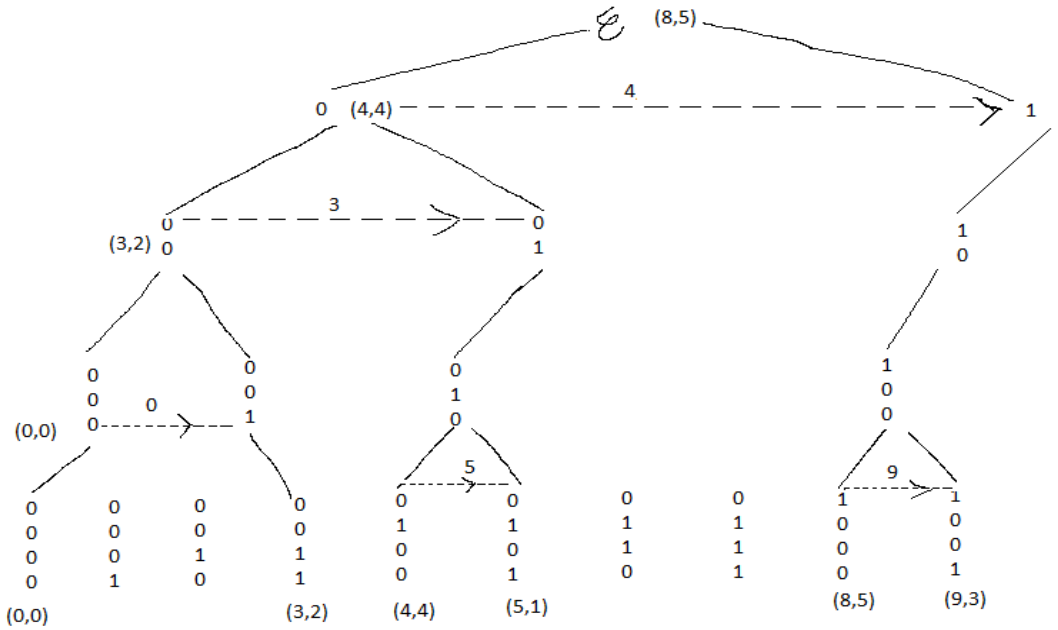

Figure 5. Linked lists are formed at the nodes

The root level processor is responsible for linking it’s child nodes into a linked list. In the Fig. 7, the root node 8 and it’s processor 5 are responsible for linking the next level node (4,4) into a linked list. The parent node will be inserted in the child linked list and will connect the tail of left child with the head of right child linked list like in figure 8.

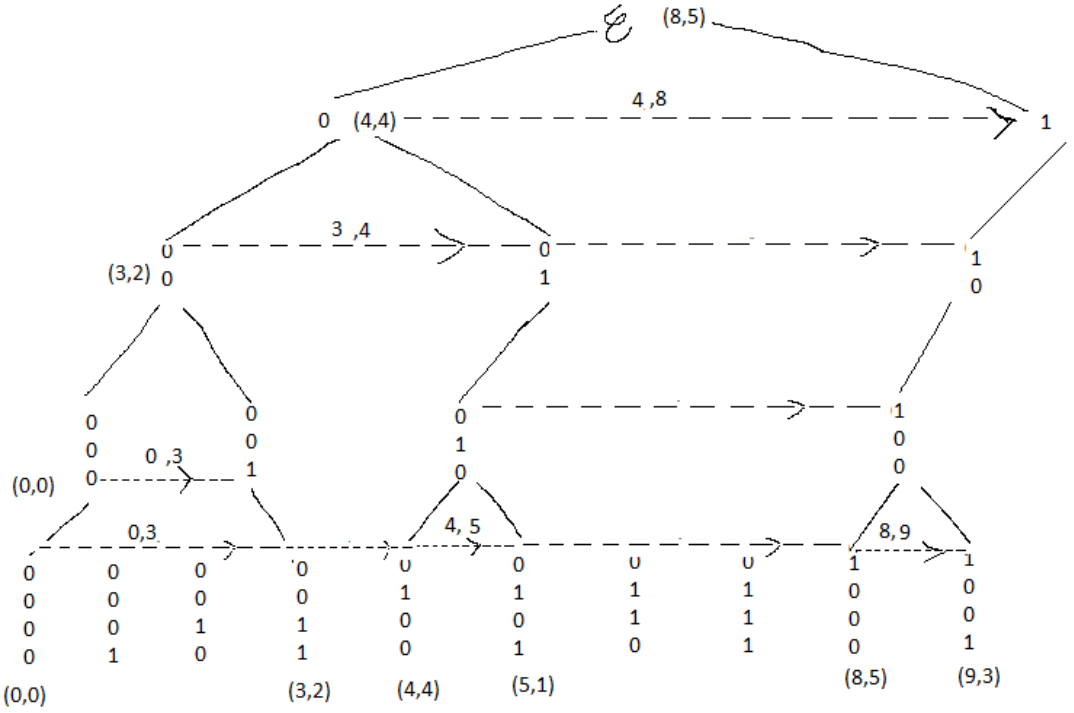

Figure 8. Parent node connects all its child linked lists

The processors of the integers in the linked list will compare the new integer with its respective number and place the new integer in it's correct position, in that way the linked list is sorted and contain all the input integers. Using nlogm processors we can form a sorted linked list in constant time with priority CRCW approach. As another approach the trie is divided into a section's, each section has (logm/t) levels and each section is given with t processors for each integer as shown in Fig. 3. Using priority concurrent read concurrent write approach with (lglgm/lgt) time the integers can be linked to a linked list in constant time.

4. Conclusion

The chaining algorithm presented in this paper has some desirable characters. An important feature demonstrated by us is that for linking them into a linked list we need not to sort the input integers into a sorted array which need at least Ω(logn/loglogn) time. In fact when nlogm processors are available n input integers in {0, 1, …, m-1} can be sorted into a linked list in constant time with Priority CRCW approach. When nt processors are available n input integers in {0, 1, …, m-1} can be sorted into a linked list in O(logm/logt) time. On the positive side the algorithm is simple and easy to program, it has no hidden factors and is fast in practical terms.

References

Beamer, P., & Histad, J. (1989). Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM, 36,* 643-670. https://doi.org/10.1145/65950.65958

Bhatt, P. C. P., Diks, K., Hagerup, T., Radzik, T., & Saxena, S. (1991). Improved Deterministic Parallel Integer Sorting. *Information and Computation, 94,* 29-47. https://doi.org/10.1016/0890-5401(91)90031-V

Fich, F. E., Ragde, P. L., & Wigderson, A. (1988). Simulations among concurrent-write PRAMs. *Algorithimica, 3,* 43-51. https://doi.org/10.1007/BF01762109

Goldberg, T., & Zwick, U. (1995) Optimal deterministic approximate parallel prefix sums and their applications. *Proceedings 1995 Israel Symposium on the Theory of Computing and Systems,* 220-228. https://doi.org/10.1007/BF01762109

Hagerup, T., & Raman, R. (1993). Fast deterministic approximate and exact parallel sorting. In *Proceedings of the 5th Annual ACM Symposium on Parallel algorithms and architectures, Velen, Germany,* 346-355. https://doi.org/10.1145/165231.157380

Hagerup, T. (1987). Towards optimal parallel bucket sorting. *Information and Computation, 75,* 39-51. https://doi.org/10.1016/0890-5401(87)90062-9

Han, Y., & Shen, X. (2002). Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. *SIAM J. Compute, 31*(6), 1852-1878. https://doi.org/10.1137/S0097539799352449

Han, Y., & Koganti, H. (2018). Searching in a Sorted Linked List. *In Proceedings of the 17th Int. Conf. on Information Technology.* https://doi.org/10.1109/ICIT.2018.00034

Karp, R. M., & Ramachandran, V. (1991) Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity,* J. van Leeuwen, Ed., New York, NY: Elsevier, 869-941. https://doi.org/10.1016/B978-0-444-88071-0.50022-9

Ragde, P. (1993). The parallel simplicity of compaction and chaining. *Journal of Algorithms, 14*(3), 371-380. https://doi.org/10.1006/jagm.1993.1019