Efficient Implementation of Tree Accumulations on Distributed-Memory Parallel Computers*

Kiminori Matsuzaki

Graduate School of Information Science and Technology, University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
kmatsu@ipl.i.u-tokyo.ac.jp

Abstract. In this paper, we develop an efficient implementation of two tree accumulations. In this implementation, we divide a binary tree based on the idea of m-bridges to obtain high locality, and represent local segments as serialized arrays to obtain high sequential performance. We furthermore develop a cost model for our implementation. The experiment results show good performance.

1 Introduction

This paper develops an efficient implementation of two *tree accumulations* for binary trees, upwards accumulation and downwards accumulation, on distributed-memory parallel computers. Tree accumulations are basic manipulations of trees: the upwards accumulation aggregates the data at all the descendants for each node, and the downwards accumulation aggregates the data at all the ancestors. These two tree accumulations have been used for solving many tree problems, for example, computing depths and sizes for all subtrees, and determining maximal independent sets [2].

For parallel tree manipulations tree contraction algorithms have been studied intensively [3,4,5,6,7]. Tree contraction algorithms are parallel algorithms that reduce a tree into the root by independent removals of nodes. Several tree contraction algorithms have been developed on many parallel computational models such as EREW PRAM [4], Hypercubes [5], and BSP/CGM [6]. Several studies have also clarified that tree accumulations can be implemented in parallel based on tree contraction algorithms [4,8].

We are developing parallel skeleton library *SkeTo* [9], which provides parallel manipulations for trees as well as lists and matrices. Compared with the implementations so far, our implementation of tree accumulations has the following three features, which are important in efficient parallel computations on distributed-memory computers.

- *High locality*. Locality is one of the most important properties in developing efficient parallel programs especially for distributed-memory computers. We adopt m-bridges [10] in the basic graph-theory to divide binary trees with high locality.
- *High sequential performance*. The performance of the sequential parts is as important as that of the communication parts for efficient parallel programs. We represent a local segment as a serialized array and implemented computations in sequential parts with loops rather than recursive functions.

* The full discussion of the paper is given in our technical report [1].

Y. Shi et al. (Eds.): ICCS 2007, Part II, LNCS 4488, pp. 609–616, 2007.

– *Cost model*. We also formalize a cost model of our parallel implementation. The cost model helps us to divide binary trees with good load balance.

The organization of the paper is as follows. In the following Sect. 2, we introduce the division and representation of binary trees in our implementation. In Sect. 3, we develop the implementation of two tree accumulations, and show the cost model. In Sect. 4, we discuss the optimal division of binary trees based on the cost model, and show experiment results in Sect. 5. In Sect. 6, we make concluding remarks.

2 Internal Representation of Binary Trees

In efficient parallel programs on distributed-memory parallel computers, we need to divide data structures into smaller parts to distribute them to the processors. Locality and load balance are the most important two properties in efficient parallel programs.

We start the discussion by introducing some graph-theoretic results [10]. Let $size_b(v)$ denote the number of nodes in the subtree rooted at vertex v.

Definition 1 (m-Critical Node [10]). *A vertex v is m-critical node, if v is an internal node and for each child v' of v inequality $\lceil size_b(v)/m \rceil > \lceil size_b(v')/m \rceil$ holds.*

The m-critical nodes divide a tree into sets of adjacent nodes (m-bridges) as shown in Fig. 1. Note that the global structure given by m-bridges also forms a binary tree.

Definition 2 (m-Bridge [10]). *An m-bridge is a set of adjacent nodes divided by m-critical nodes, that is, a largest set of adjacent nodes in which m-critical nodes are only on the ends.*

Let N be the number of nodes and P be the number of processors. In the previous studies [10,11], we divided a tree into m-bridges using m given by $m = 2N/P$. Under this division, we obtain at most $(2P-1)$ m-bridges and each processor deals with at most two m-bridges. Of course this division enjoys high locality, but it has poor load balance since the maximum number of nodes passed to a processor may be $2N/P$, which is twice of that in the best load-balancing case. In this paper, we divide a tree into m-bridges using smaller m and assigns more m-bridges to a processor. In Sect. 4, we discuss how to adjust m based on the cost model of our implementation.

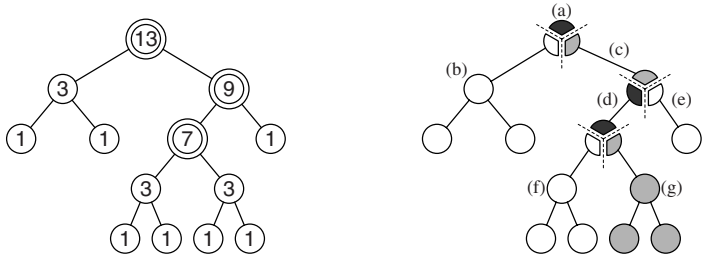
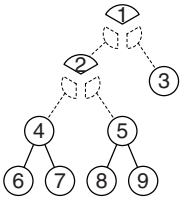


Fig. 1. An example of m-critical nodes and m-bridges. Left: There are three 4-critical nodes denoted by the doubly-lined circles. The number in each node denotes the number of nodes in the subtree. Right: There are seven 4-bridges, (a)–(g), each of which is a set of connected nodes.



$$gt = [\square^{\mathsf{N}}, \square^{\mathsf{N}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}]$$
$$seg = [\,[\,1^{\mathsf{C}}\,],\ [\,2^{\mathsf{C}}\,],\ [\,4^{\mathsf{N}}, 6^{\mathsf{L}}, 7^{\mathsf{L}}\,],\ [\,5^{\mathsf{N}}, 8^{\mathsf{L}}, 9^{\mathsf{L}}\,],\ [\,3^{\mathsf{L}}\,]\,]$$

Fig. 2. Internal representation of divided binary trees. Each local segment is distributed to one of processors and is not shared. Labels L, N and C denote a leaf, a normal internal node, and a critical node, respectively. Each critical node is included in the parent segment.

Generally speaking, a tree structure is often implemented with pointers or references. In this paper, we represent the tree structure with arrays as shown in Fig. 2 where the elements are aligned in the order of preorder traversal. This representation has an advantage in terms of locality: that is, we can reduce cache misses since adjoining elements are aligned one next to another.

In the following, gt denotes the serialized array for the global structure, and $seg[i]$ denotes the serialized array for the ith local segment. We use functions isLeaf, isNode and isCrit to check whether the node is a leaf, an internal node, and a critical node.

3 Implementation and Cost Model of Tree Accumulations

In this section, we show the implementation and the cost model of the tree accumulations on distributed-memory parallel computers. We implement the local computations in tree accumulations using loops and stacks on the serialized arrays. This is the most significant contribution in this paper with which the parallel programs reduce the cache misses and achieve good speedups even against efficient sequential programs.

Before showing two accumulations, we introduce several parameters for the cost model. The computational time of function f executed with p processors is denoted by $t_p(f)$. Parameter N denotes the number of nodes, and P denotes the number of processors. Parameter m is used for m-critical nodes and m-bridges, and M denotes the number of segments after the division. For the ith segment, in addition to the parameter of the number of nodes L_i, we introduce parameter D_i indicating the depth of the critical node. Parameter c_α denotes the communication time for a value of type α. We develop the cost model with the overall communication cost.

The cost model for tree accumulations can be uniformly given as the sum of the maximum cost of local computations and the cost of global computations as follows.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m$$

where $\sum_{pr(i)=p}$ denotes the summation of costs for m-bridges associated to processor p, and t_l, t_d, and t_m are given with the cost of functions and communications. The term $(L_i \times t_l)$ indicates the computational time required in sequential computation and the term $(D_i \times t_d)$ indicates the overhead caused by parallelism. The last term $(M \times t_m)$ indicates the overhead in terms of global structure.

Upwards Accumulation. Upwards accumulation applies a ternary function to each internal node in a bottom-up manner and returns a tree of the same shape as the input. In Haskell notation, a sequential definition of the upwards accumulation is given as follows.

$$\begin{aligned}
&\mathsf{uAcc} :: (\alpha \to \beta \to \alpha \to \alpha) \to \mathit{Tree}\ \alpha\ \beta \to \mathit{Tree}\ \alpha\ \alpha \\
&\mathsf{uAcc}\ k\ (\mathit{Leaf}\ a) \quad\ = \mathit{Leaf}\ a \\
&\mathsf{uAcc}\ k\ (\mathit{Node}\ l\ b\ r) = \mathbf{let}\ l' = \mathsf{uAcc}\ k\ l; r' = \mathsf{uAcc}\ k\ r; \\
&\qquad\qquad\qquad\qquad\ \mathbf{in}\ \mathit{Node}\ l'\ (k\ (\mathit{root}\ l)\ b\ (\mathit{root}\ r))\ r'
\end{aligned}$$

Function *root* returns the root node of the given tree. The upwards accumulation takes $(N \times t_1(k)/2)$ time by sequential execution.

For efficient implementation of the upwards accumulation, we require four auxiliary functions ϕ, ψ_n, ψ_l, and ψ_r satisfying the following equations.

$$\begin{aligned}
&k\ l\ b\ r = \psi_n\ l\ (\phi\ b)\ r \\
&\psi_n\ (\psi_n\ x\ l\ y)\ b\ r = \psi_n\ x\ (\psi_l\ l\ b\ r)\ y \\
&\psi_n\ l\ b\ (\psi_n\ x\ r\ y) = \psi_n\ x\ (\psi_r\ l\ b\ r)\ y
\end{aligned}$$

Let the type of result of ϕ be γ. Intuitive meaning of these auxiliary functions is as follows: The computation on an internal node is lifted up by function ϕ to some domain and pulled down by function ψ_n from the domain, where a certain kind of associativity holds against functions ψ_l and ψ_r on the domain.

Under this condition, we implement the upwards accumulation by five steps.

At the first step, we apply the following UACC_LOCAL to each segment to compute local upwards accumulation. This function puts the intermediate result to array seg' if a node has no terminal node as descendants. (This result value is indeed the result of uAcc.) This function returns the result of the local reduction and the array seg'.

```
UACC_LOCAL(k, φ, ψ_l, ψ_r, seg)
  stack ← ∅; d ← −∞;
  for i ← seg.size − 1 to 0
    if (isLeaf(seg[i])) seg'[i] ← seg[i]; stack ← seg'[i]; d ← d + 1;
    if (isNode(seg[i]))
      lv ← stack; rv ← stack;
      if (d == 0) stack ← ψ_l(lv, φ(seg[i]), rv); d ← 0;
      else if (d == 1) stack ← ψ_r(lv, φ(seg[i]), rv); d ← 0;
      else seg'[i] ← k(seg[i], lv, rv); stack ← seg'[i]; d ← d − 1;
    if (isCrit(seg[i])) stack ← φ(seg[i]); d ← 0;
  top ← stack; return(top, seg');
```

In the computation of UACC_LOCAL, ϕ and either of ψ_l or ψ_r are applied to each node on the path from the critical node to the root, and k is applied to the other internal nodes. Since the number of internal nodes is a half of L_i, we obtain the cost of UACC_LOCAL as $\max_p \sum_{pr(i)=p}((L_i/2 - D_i) \times t_1(k) + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))))$.

At the second step, we gather the results of the local reductions to the global structure gt on the root processor. From each leaf segment a value of type α is sent, and from each internal segment a value of type γ is sent. Therefore, the communication cost in the second step is given as $M/2 \times c_\alpha + M/2 \times c_\gamma$.

At the third step, we compute the upwards accumulation for the global structure gt on the root processor. Function UACC_GLOBAL performs sequential upwards accumulation using function ψ_n.

```
UACC_GLOBAL(ψ_n, gt)
    stack ← ∅;
    for i ← gt.size − 1 to 0
        if (isLeaf(gt[i])) gt'[i] ← gt[i];
        if (isNode(seg[i])) lv ← stack; rv ← stack; gt'[i] ← ψ_n(lv, gt[i], rv);
        stack ← gt'[i];
    return(gt');
```

In this function, we apply function ψ_n to each internal segment of gt, and thus the cost of the third step is given as $M/2 \times t_1(\psi_n)$.

At the fourth step, we send the results of global upwards accumulation to processors, where two values are sent to each internal segment and no value is sent to each leaf segment. All the values have type α after the global upwards accumulation, and thus the communication cost in the fourth step is given as $M \times c_\alpha$.

At the last step, we apply function UACC_UPDATE to each internal segment. The two values pushed to the stack are the values passed in the previous step, and correspond to the two results of children of the critical node. Note that in the last step we only compute the missing values left in the segment seg'.

```
UACC_UPDATE(k, seg, seg', lc, rc)
    stack ← ∅; stack ← rc; stack ← lc; d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i])) stack ← seg'[i]; d ← d + 1;
        if (isNode(seg[i]) ∨ isCrit(seg[i]))
            lv ← stack; rv ← stack;
            if (d == 0 ∨ d == 1 ∨ isCrit(seg[i]))
                seg'[i] ← k(seg[i], lv, rv); stack ← seg'[i]; d ← 0;
            else stack ← seg'[i]; d ← d − 1;
    return(seg');
```

In this step, function k is applied to the nodes on the path from the critical node to the root. Noting that the depth of the critical nodes in ith segment is D_i, we can give the cost as $\max_p \sum_{pr(i)=p}(D_i \times t_1(k))$.

Summarizing the discussion above we can give the cost model of the upwards accumulation as follows. Note that the coefficient of L_i is the same as that of sequential programs and is not affected by the introduction of auxiliary functions.

$$t_p(\mathsf{uAcc}\ k) = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) + M \times (3c_\alpha + c_\gamma + t_1(\psi_n))/2$$

Downwards Accumulation. We then show the implementation of the downwards accumulation. Downwards accumulation is a top-down computation where an accumulative parameter c is updated with two functions (g_l, g_r). The values of leaves are not used in this computation.

$$\begin{aligned}
&\mathsf{dAcc} :: (\gamma \to \beta \to \gamma, \gamma \to \beta \to \gamma) \to \gamma \to \mathit{Tree}\ \alpha\ \beta \to \mathit{Tree}\ \gamma\ \gamma \\
&\mathsf{dAcc}\ (g_l, g_r)\ c\ (\mathit{Leaf}\ a) \quad = \mathit{Leaf}\ c \\
&\mathsf{dAcc}\ (g_l, g_r)\ c\ (\mathit{Node}\ l\ b\ r) = \mathbf{let}\ l' = \mathsf{dAcc}\ (g_l, g_r)\ (g_l\ c\ b)\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad r' = \mathsf{dAcc}\ (g_l, g_r)\ (g_r\ c\ b)\ r \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{Node}\ l'\ c\ r'
\end{aligned}$$

The downwards accumulation takes $(N \times (t_1(g_l) + t_1(g_r))/2)$ by sequential execution.

For efficient parallel implementation, we require auxiliary functions ϕ_l, ϕ_r, ψ_u and ψ_d satisfying the following equations. Let δ be the type of results of ϕ_l and ϕ_r.

$$g_l\ c\ n = \psi_d\ c\ (\phi_l\ n), \quad g_r\ c\ n = \psi_d\ c\ (\phi_r\ n), \quad \psi_d\ (\psi_d\ c\ n)\ m = \psi_d\ c\ (\psi_u\ n\ m)$$

The implementation of the downwards accumulation also consists of five steps. Due to the page limit, we only show the outline of the implementation. See [1] for details.

1. For each internal segment, compute intermediate values corresponding to the path from the root to the critical node. We can implement this step by a reversed loop with a stack. The cost is $\max_p \sum_{pr(i)=p}(D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)))$.
2. Gather local results to the root processor. The communication cost is $M \times c_\delta$.
3. Compute downwards accumulation on the root processor for the global structure by a forward loop with a stack. The cost of this step is given as $M \times t_1(\psi_d)$.
4. Distribute the result of global downwards accumulation to each segment. The communication cost in this step is $M \times c_\gamma$.
5. For each segment, compute downwards accumulation starting from the result of global downwards accumulation. We can implement this computation using a forward loop with a stack. The cost of this step is given as $\max_p \sum_{pr(i)=p}(L_i/2 \times (t_1(g_l) + t_1(g_r)))$.

The overall cost model for the downwards accumulation is given as follows. Here again the coefficient of L_i is the same as that of sequential computation.

$$\max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(g_l) + t_1(g_r)}{2} + D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \right) + M \times (c_\delta + t_1(\psi_d) + c_\gamma)$$

4 Optimal Division of Binary Trees Based on Cost Model

Locality and load balance are two major properties in developing efficient parallel programs. In the division of trees, we enjoy good locality with smaller m while we enjoy good load balance with larger m. Therefore we need to find an appropriate value for m.

The parameters L_i, D_i, and M in the cost model satisfy the following inequalities against m:

$$L_i \leq m, \quad D_i \leq L_i/2 \leq m/2, \quad (N/m - 1)/2 < M < 2N/m - 1.$$

By using a greedy balancing algorithm with respect to the number of nodes and depth of the critical node for distributing m-bridges, we get the cost of worst case as follows.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \leq (N/P + m) \times (t_l + t_d/2) + M \times t_m$$

The equality holds if all the m-bridges have the same number of nodes and all the critical nodes are in depth $m/2$, i.e., fully ill-balanced trees. With the inequality between M and m, the right-hand side above gets the minimum value for some value m in the following range.

$$\sqrt{t_m/(2t_l + t_d)}\sqrt{N} < m < 2\sqrt{t_m/(2t_l + t_d)}\sqrt{N}$$

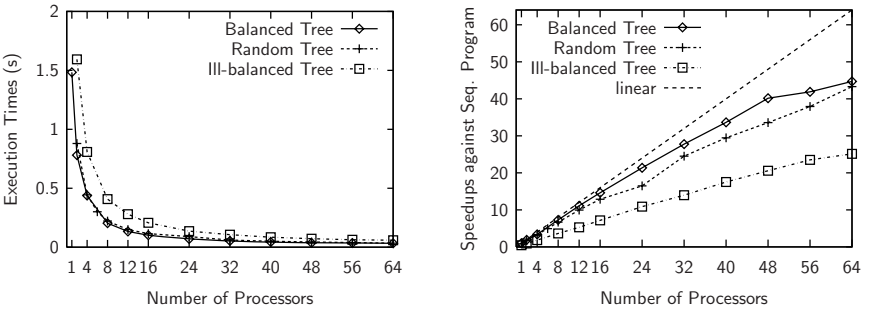


Fig. 3. Execution times and speedups against sequential program where $m = 2 \times 10^4$

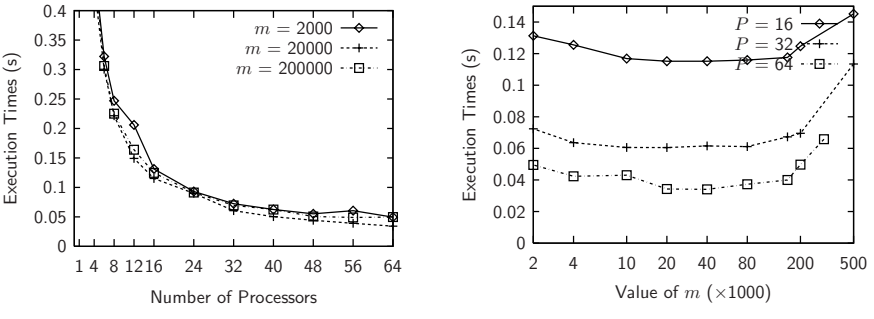


Fig. 4. Execution times changing parameter m for the randomly generated tree

5 Experiment Results

To verify the efficiency of the implementation of tree accumulations, we made several experiments. We used our PC-cluster of uniform PCs with Pentium 4 2.8 GHz CPU and 2 GByte memory connected with Gigabit Ethernet. The compiler and MPI library used are gcc 4.1.1 and MPICH 1.2.7, respectively.

We solved the party planning problem [12], which is a generalization of maximal independent sets [2]. The input trees are (1) a balanced tree, (2) a randomly generated tree and (3) a fully ill-balanced tree, each with 16777215 ($= 2^{24} - 1$) nodes. The parameters for the cost model are $t_l = 0.18$ μs, $t_d = 0.25$ μs, and $t_m = 100$ μs on our PC cluster.

Figure 3 shows the general performance of the tree accumulations. Each execution time excludes the initial data distribution and final gathering. The speedups are plotted against a sequential implementation of the tree accumulations. As seen in these plots, the implementation shows not only scalability but also good sequential performance. For the fully ill-balanced tree the implementation performs worse but this is caused by the factor of $D_i \times t_d$ ($\sim 0.7(L_i \times t_l)$, for the program) introduced for parallelism.

To analyze more in detail, we made more experiments by changing the value of m. The results are shown in Fig. 4. Roughly speaking, as seen from Fig. 4 (left), the implementation of tree accumulations scales under both large and small m. Figure 4 (right) plots the execution time with respect to the parameter m. The performance gets

worse for too small m or too large m, where good performance is shown under the range $5 \times 10^4 < m < 1 \times 10^5$ computed from the parameters of the cost model estimated with a small test case.

6 Conclusion

We have developed an efficient implementation of the tree accumulations. Not only our implementation shows good performance against efficient sequential programs, but also the cost model of the implementation helps us to divide a tree into segments. The implementation will be available as part of SkeTo library (Sketo library is available at http://www.ipl.t.u-tokyo.ac.jp/sketo/). Our future work is to develop a profiling system that determines accurate parameter m for dividing trees.

Acknowledgments. This work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) 17300005, and the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) 18700021.

References

1. Matsuzaki, K., Hu, Z.: Efficient implementation of tree skeletons on distributed-memory parallel computers. Technical Report METR 2006-65, Department of Mathematical Informatics, the University of Tokyo (2006).
2. He, X.: Efficient parallel algorithms for solving some tree problems. In 24th Allerton Conference on Communication, Control and Computing. (1986).
3. Miller, G.L., Reif, J.H.: Parallel tree contraction and its application. In 26th Annual Symposium on Foundations of Computer Science, 21–23 October 1985, Portland, Oregon, USA, IEEE Computer Society (1985).
4. Abrahamson, K.R., Dadoun, N., Kirkpatrick, D.G., Przytycka, T.M.: A simple parallel tree contraction algorithm. Journal of Algorithms **10**(2) (1989).
5. Mayr, E.W., Werchner, R.: Optimal tree contraction and term matching on the hypercube and related networks. Algorithmica **18**(3) (1997).
6. Dehne, F.K.H.A., Ferreira, A., Cáceres, E., Song, S.W., Roncato, A.: Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. Algorithmica **33**(2) (2002).
7. Vishkin, U.: A no-busy-wait balanced tree parallel algorithmic paradigm. In SPAA 2000: Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures, July 9–13, 2000, Bar Harbor, Maine, USA, ACM Press (2000).
8. Gibbons, J., Cai, W., Skillicorn, D.B.: Efficient parallel algorithms for tree accumulations. Science of Computer Programming **23**(1) (1994).
9. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In InfoScale '06: Proceedings of the 1st international conference on Scalable information systems. Volume 152 of ACM International Conference Proceeding Series., ACM Press (2006).
10. Reif, J.H., ed.: Synthesis of Parallel Algorithms. Morgan Kaufmann Publishers (1993).
11. Matsuzaki, K., Hu, Z., Takeichi, M.: Implementation of parallel tree skeletons on distributed systems. In The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 – December 1, 2002, Proceedings. (2002).
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. Second ed. MIT Press (2001).