Parallel Algorithm for Finding the Minimum Edges to Build a Strongly Connected Directed Graph

AKIO TADA
Sojo University
Faculty of Computer and Information Sciences
Department of Computer System Technology
Ikeda 4-22-1, Kumamoto City
JAPAN

EIICHI MUKAI
Sojo University
Faculty of Computer and Information Science
Department of Computer System Technology
Ikeda 4-22-1, Kumamoto City
JAPAN

MASAHIRO MIGITA
Kumamoto University
Center for Multimedia and Information Technologies
Kurokami 2-39-1, Kumamoto City
JAPAN

TSUYOSHI ITOKAWA
Kumamoto University
Graduate School of Science and Technology
Computer Science and Electrical Engineering
Kurokami 2-39-1, Kumamoto City
JAPAN

Abstract: The problem of finding the minimum edges to build a strongly connected directed graph is one of the most fundamental problems in graph theory. The known parallel algorithm solves this problem in $O(\log n)$ time using $O(n^3)$ processors on a CRCW PRAM model. In this paper, we propose a parallel algorithm to find the minimum edges to build a strongly connected directed graph for a disconnected directed acyclic graph in $O(\log(n+m))$ time using $O(n + m)$ processors on a CREW PRAM model. This algorithm is an efficient parallel algorithm because the number of processors depends on the density of the given graph and the time complexity is also more efficient when compared with the identical model.

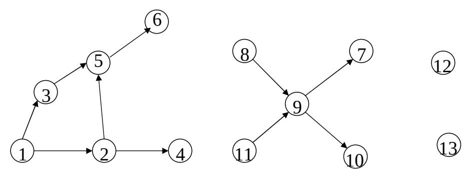
Key–Words: Parallel algorithm, Minimum edges, Directed acyclic graph(DAG), Strongly connected component(SCC), CREW PRAM model

1 Introduction

The problem of finding the minimum edges to make a disconnected directed graph strongly connected is one of the fundamental problems in graph theory. The known parallel algorithm[1] in a disconnected directed graph with n vertices and m edges takes $O(\log n)$ time using $O(n^3)$ processors on a CRCW PRAM model. Since this algorithm depends on the transitive closure matrix calculation, it is difficult to reduce the number of processors furthermore.

In this paper, we propose an efficient parallel algorithm for finding the minimum edges to build a strongly connected directed graph for a disconnected directed acyclic graph on a CREW PRAM model, using only the basic parallel algorithms. Namely, the proposed algorithm initially finds source and sink vertices in a given graph, and divides the given graph into several linked lists, in which each vertex has at most 1 both input and output edge degree. Next we detect each connected component on the divided linked lists, and connect from a sink to a source between each connected component. Finally, after the remaining unlinked sources and sinks are linked, the given graph is made into one strongly connected component graph. This algorithm requires $O(\log(n + m))$ time and $O(n + m)$ processors on a CREW PRAM model.

This algorithm is an efficient parallel algorithm because it requires at most $O(n^2)$ processors if the given graph is a dense graph ($m = O(n^2)$) and also requires only $O(n)$ processors if the given graph is a sparse graph ($m = O(n)$). Moreover, the time complexity of the known algorithm is $O(\log n)$ on a CRCW PRAM model, while that of the proposed algorithm is $O(\log(n+m))$ on a CREW PRAM model, therefore the proposed algorithm is more efficient than the known algorithm when compared with the identical model.



	1	2	3	4	5	6	7	8	9	10
OV	1	1	2	2	3	5	8	9	9	11
IV	2	3	4	5	5	6	9	7	10	9

Figure 1: Example of given disconnected directed graph and array representation of edges.

2 The Proposed Parallel Algorithm

A given graph is a disconnected directed acyclic graph, which contains isolated vertices but does not contain cycles and multiple edges. Each vertex of connected components in a given graph is serially numbered. The number of minimum added directed edges to make a disconnected directed graph strongly connected is given by Theorem 1 in [1] as follows:

Theorem 1 *At least max(n_{so}, n_{si}) + n_{is} edges are needed to make a given graph strongly connected, where n_{so} is the number of source vertices, n_{si} is the number of sink vertices and n_{is} is the number of isolated vertices.*

Namely,

$$minimum\_edges = max(n_{so}, n_{si}) + n_{is}$$

The proposed parallel algorithm is composed of the following two stages:

- Stage 1: Divide a disconnected directed graph into several linked lists
- Stage 2: Connect each source and sink to make a disconnected graph strongly connected

A disconnected directed graph with n vertices and m edges is given in the form of two arrays of directed edges and the number of isolated vertices. Let the array OV[$1..m$] be the out-vertex numbers and the array IV[$1..m$] be the in-vertex numbers. Each i-th directed edge is represented by a pair of OV[i] and IV[i], and the edges are sorted in order of the out-vertex number. Isolated vertices do not appear in the input array and they are assigned their vertex numbers larger than those used in connected components. An example of a disconnected directed acyclic graph and its array representation of directed edges is shown in Figure 1.

In the following subsections the details of Stages 1 and 2 are respectively described. Stage 1 of this algorithm is similar to the Stage 1 in [2, 3].

2.1 Stage 1: Divide a disconnected directed graph into several linked lists

In this stage, we initially find sources and sinks in the given graph and they are counted. Next, the given graph is divided into several linked lists, in which each vertex has at most 1 both input and output degree. The list number is also assigned to each linked list, according to the number of a heading vertex of each list, and they are serially reassigned from 1. The consecutive list numbers are assigned in each connected component because the consecutive vertex numbers are given in each connected component. Finally, the sources and sinks of the linked lists are stored in two arrays. This stage is composed of the following three steps.

Step 1.1 The in-degree and the out-degree for each vertex are calculated. As the vertex with the in-degree = 0 is a source and the vertex with the out-degree = 0 is a sink, the sinks and sources in the given graph are counted. Let the bigger number of the in-degree and out-degree be the maximum degree for each vertex. Each vertex which has a maximum degree greater than 1 will be divided into some new vertices. We call an original vertex in the given graph a 'divided vertex' and call the new generated vertices 'sibling vertices' of each divided vertex. Therefore, the number of sibling vertices generated from a divided vertex is equal to the maximum degree. This step requires $O(\log m)$ time and $O(m)$ processors.

Step 1.2 The sibling vertices of each divided vertex are serially numbered to keep the same order of the vertex number in the given graph. Here a correspondence table N[1..S[n]] which corresponds the new vertex number to the old vertex number is prepared, where S[n] is the total number of all new vertices after division and S[n] = $O(n + m)$. Subsequently, the old vertex numbers in OV[$1..m$] and IV[$1..m$] are substituted for the new vertex numbers, and new edges arrays NOV[$1..m$] and NIV[$1..m$] are created respectively. After all, the given graph is divided into several linked lists with new vertex numbers. This step requires $O(\log m)$ time and $O(n + m)$ processors.

In the example, Figure 2 shows the correspondence table N and the new edges arrays NOV and NIV.

Step 1.3 A list number is assigned to each linked list. Initially, the heading vertex number of each linked list is broadcasted[5], and the list numbers stored in an array LISTN[1..S[n]] are arranged in

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
N	1	1	2	2	3	4	5	5	6	7	8	9	9	10	11	12	13

	1	2	3	4	5	6	7	8	9	10
NOV	1	2	3	4	5	7	11	12	13	15
NIV	3	5	6	8	7	9	12	10	14	13

⑯ ⑰

Figure 2: Correspondence table N and new edges arrays NOV and NIV.

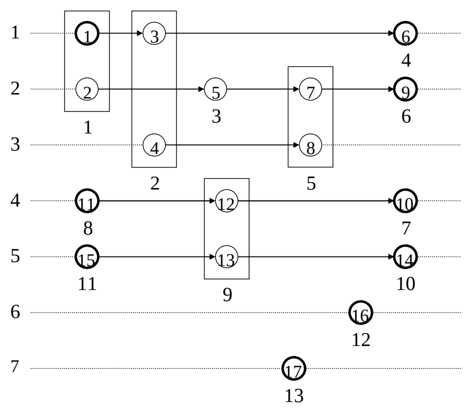
ascending order by Parallel Merge Sort[6]. Then the list numbers are serially renumbered from 1. Next, we make a table of a source and a sink on each linked list. Namely, the old vertex numbers of a source and sink on each linked list are stored in the arrays SO[1..L] and SI[1..L] respectively. Here we let L be the maximum number of linked lists, and $L = O(n+m)$ because each linked list has at least one new vertex. If sources or sinks are sibling vertices, then only the heading sibling old vertex number is stored. This step requires $O(\log(n+m))$ time and $O(n+m)$ processors because the list numbers of new vertices are sorted.

Therefore, the complexity of Stage 1 requires $O(\log(n+m))$ time and $O(n+m)$ processors.

As the result of Stage 1, the divided linked lists and the arrays SO and SI are illustrated as Figure 3. In this figure, the given graph is divided into 7 linked lists. The number included in each circle is its new vertex number, the squares represent the sibling vertices, the subscript of each vertex is the old vertex number and the numbers on the left side are list numbers.

2.2 Stage 2: Connect each source and sink to make a disconnected graph strongly connected

In the divided linked lists, each connected component is composed of more than one linked list, and has at least one source and sink. In this stage, initially the boundary linked lists which separate connected components are found, and at each boundary linked list, a sink in each connected component and a source in the next connected component are linked by searching the table of arrays SO and SI. Subsequently, the given disconnected components are made into one connected component. Next, the sink of the linked list with the maximum list number and the source of the linked list with the minimum list number are connected. At this time, the given graph has one strongly connected com-



	1	2	3	4	5	6	7
SO	1	0	0	8	11	12	13
SI	4	6	0	7	10	12	13

Figure 3: Divided linked lists and arrays SO and SI.

ponent. Finally the remaining unlinked sources and sinks are linked with the minimum directed edges. At the end of this stage, the given disconnected directed graph is made into one strongly connected component graph. This stage is composed of the following three steps.

Step 2.1 The boundary linked lists which separate connected components are found. Initially, the maximum list number of sibling vertices is calculated by the partial maximum parallel algorithm[4] on an array LISTN[1..S[n]]. Then the indivisual maximum list number of each linked list is found by the doubling technique[4]. Finally, the maximum list number of each linked list is corrected, so as to be not decreasing along the number of each linked list. Then, if each maximum list number is equal to its list number, its linked list is separate between one connected component and the next connected component, and it is marked. This step requires $O(\log(n+m))$ time and $O(n+m)$ processors.

Step 2.2 All connected components in the given disconnected graph are linked according to the list number, and made into one connected component. We add each directed edge, which links from a sink in a connected component to a source in the next connected component on each boundary found in Step 2.1. Here there is a case where the boundary linked list does not have a sink, so the array SI must be searched by the doubling

	1	2	$\textcircled{3}$	4	$\textcircled{5}$	$\textcircled{6}$	$\textcircled{7}$
AEO	0	0	6	0	10	12	13
AEI	0	0	8	0	12	13	1

Figure 4: Added edges in Step2.2.

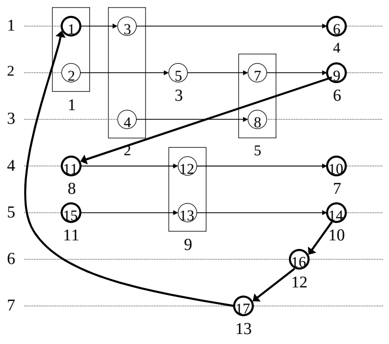


Figure 5: Connected graph.

	1	2	3	4	5	6	7
SO	0	0	0	0	11	0	0
SI	4	0	0	7	0	0	0

Figure 6: Remaining unlinked sources and sinks.

	1	2
SO	11	0
SI	4	7
(INDEX)	(1)	(4)

$\downarrow$

	1	2
SO	11	1
SI	4	7
(INDEX)	(1)	(4)

$\swarrow \quad \downarrow$

	1	2	3	4	5	6	7
AEO	4	0	6	7	10	12	13
AEI	11	0	8	1	12	13	1

Figure 7: New added edges.

technique to find a sink with the nearest list number in the same connected component. Similarly, there is a case where the next linked list of the boundary linked list does not have a source, so the array SO must be searched as the same as a sink. Next, an edge which links from the sink with the largest list number to the source with the smallest list number is added. At this time, the given graph has one strongly connected component. The arrays AEI[$1..L$] and AEO[$1..L$] are used to store the in-vertices and out-vertices on the added directed edges, and already linked sources and sinks are deleted from arrays SO and SI respectively. Step 2.2 requires $O(\log(n+m))$ time and $O(n+m)$ processors.

In the example, Figure 4 shows the arrays AEO and AEI representing the added edges in this step. The marked index numbers represent the boundaries of connected components found in Step 2.1. Figure 5 also shows the connected graph.

Step 2.3 The minimum directed edges are added to link from the remaining unlinked sinks to the remaining unlinked sources, using the arrays SO and SI. Initially, the vertex numbers with index numbers in the arrays SO and SI are sorted respectively to get together the remaining unlinked sources and sinks. In order to make the numbers of sources and sinks even, if the number of sources is smaller than the number of sinks, fill the array SO with the vertex number of the source which has the smallest list number found in Step 2.2, or vice versa. Then a pair of each directed edge in the arrays SO and SI is stored to the arrays AEO and AEI according to the specific index number(INDEX). Finally we obtain one strongly connected component graph by adding the directed edges in the arrays AEO and AEI to the given disconnected directed graph. Step 2.3 requires $O(\log(n+m))$ time and $O(n+m)$ processors because the arrays SO and SI must be sorted.

In the example, Figure 6 shows the remaining unlinked sources and sinks. Figure 7 shows the result of sorting and linking the remaining sources and sinks, and also the new added edges.

Stage 2 requires $O(\log(n+m))$ time and $O(n+m)$ processors.

Now, we will calculate the number of the added edges in this stage. Let n_{cc} be the number of the given connected components. Again, let n_{so}, n_{si} and n_{is} be the number of sources, sinks and isolated vertices respectively. Then, in Step 2.2 the number of the added edges is (n_{cc} + n_{is} - 1) + 1, and in Step 2.3 it is max(n_{so} - n_{cc}, n_{si} - n_{cc}). Namely,

$$\begin{aligned} added\_edges &= (n_{cc}+n_{is}-1)+1 \\ &\quad + max(n_{so}-n_{cc}, n_{si}-n_{cc}) \\ &= max(n_{so}, n_{si}) + n_{is} \end{aligned}$$

Accordingly, this expression satisfies Theorem 1.

Finally, we show the final result of the strongly connected graph constructed by this parallel algorithm

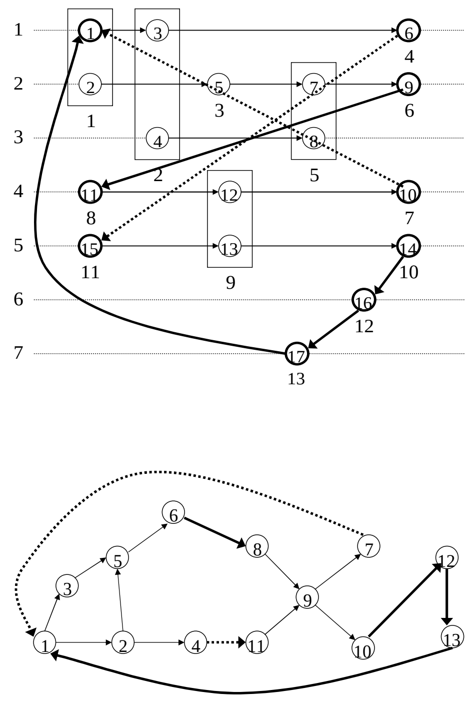


Figure 8: Constructed strongly connected component graph.

in Figure 8. The bold lines are the edges added in Step 2.2 and the dotted bold lines are those added in Step 2.3. In Figure 8, the given graph of the example has 3 sources, 4 sinks and 2 isolated vertices, then the number of the new added edges is 6. So 4+2=6 edges satisfy Theorem 1.

3 Conclusion

We have proposed a parallel algorithm for finding the minimum edges to make a disconnected directed acyclic graph strongly connected. This parallel algorithm requires $O(\log(n + m))$ time and $O(n + m)$ processors On a CREW PRAM model.

This algorithm is an efficient parallel algorithm because it requires at most $O(n^2)$ processors if the given graph is a dense graph $(m = O(n^2))$ and also requires only $O(n)$ processors if the given graph is a sparse graph $(m = O(n))$. Moreover, the time complexity of the known algorithm is $O(\log n)$ on a CRCW PRAM model, while that of the proposed algorithm is $O(\log(n + m))$ on a CREW PRAM model, therefore when compared with the identical model, the proposed algorithm is more efficient than the known algorithm.

References:

[1] Chandhui, P.: An O(log n) Parallel Algorithm for Strong Connectivity Augmentation Problem, $Int.J.Comput.Math.$, Vol.22,pp.187-197 (1987).

[2] Tada, A., Migita M. and Nakamura, R. ”Parallel Topological Sorting Algorithm,” (in Japanese) IPSJ Journal, Vol.45, No.4, pp.1102-1111 (2004).

[3] Migita, M., Tada, A., Itokawa, T. and Nakamura, R. ”Parallel Algorithm for Determining Critical Paths in PERT Chart,” (in Japanese) IPSJ Journal, Vol.47, No.7, pp.2212-2223 (2006).

[4] Gibbons A. and Rytter, W.: Efficient parallel algorithm, pp.6-18, Cambridge University Press, Cambridge (1988).

[5] Xavier, C. and Iyegar, S.S.: Introduction to Parallel Algorithms, pp.108-140, Wiley-inter science (1998).

[6] Cole, R. ”Parallel Merge Sort,” SIAM J. Comput., Vol.17, No.4, pp.770-785 (1988).