New Parallel Shortest Path Searching Algorithm based on Dynamically Reconfigurable Processor DAPDNA-2

Hiroyuki ISHIKAWA, Sho SHIMIZU, Yutaka ARAKAWA, Naoaki YAMANAKA, Kosuke SHIBA
Department of Information and Computer Science, Faculty of Science and Technology, Keio University, 3–14–1 Hiyoshi, Kohoku-ku, Yokohama, 223-8522, Japan
IPFlex Inc. Kamiosaki 2–27–1, Sun felista Meguro 6F, Shinagawa-ku, Tokyo, 141–0021 Japan
Email: ishikawa@yamanaka.ics.keio.ac.jp

Abstract—This paper proposes a parallel shortest path-searching algorithm and implements it on a newly structured parallel reconfigurable processor, DAPDNA-2 (IPFlex Inc). Routing determines the shortest paths from the source to the ultimate destination through intermediate nodes.

In Open Shortest Path First (OSPF), Dijkstra’s shortest path algorithm, which is the conventional one, finds the shortest paths from the source on a program counter-based processor. The calculation time for Dijkstra’s algorithm is \(O(N^2)\) when the number of nodes is \(N\). When the network scale is large, calculation time required by Dijkstra’s algorithm increases rapidly. It’s very difficult to compute Dijkstra’s algorithm in parallel because of the need for previous calculation results, so Dijkstra’s algorithm is unsuitable for parallel processors.

Our proposed scheme finds the shortest paths using a simultaneous multi-path search method. In contrast with Dijkstra’s algorithm, several nodes can be determined at one time. Moreover, we partition the network into different groups (network groups) and find the all-node pair’s shortest path in each group using a pipeline operation. Networks can be abstracted, and the shortest paths in very large networks can be found easily. The proposed scheme can decrease calculation time from \(O(N^2)\) to \(O(N)\) using a pipeline operation on DAPDNA-2. Our simulations show that the proposed algorithm uses 99.6% less calculation time than Dijkstra’s algorithm. The proposed algorithm can be applied to the very large Internet network designs of the future.

I. INTRODUCTION

In today’s Internet, routing protocols become increasingly important technologies. Each link is associated with a cost and routers exchange link state information to have a complete description of the network topology. For OSPF[1], [2], [3], which is a link state based routing protocol, each router computes a path with minimum cost from itself to each other router in the area. A routing table can be constructed on the shortest path tree (SPT). Dijkstra’s algorithm[4] finds the shortest paths from the source in OSPF.

The calculation time of Dijkstra’s shortest path algorithm increases rapidly as network scale increases, so a high-end CPU and memory are required to use it. Recently, there has been an increase in research on routing based on several kinds of metrics such as delay or network bandwidth. However, it takes a long time to find the shortest paths using the program counter-based processor.

In this research, we describe a parallel-searching shortest path algorithm and implement the algorithm on DAPDNA-2, which is a dynamically reconfigurable processor developed by IPFlex Inc[5]. DAPDNA-2 consists of DAP (Digital Application Processor), a high-performance RISC core and DNA (Digital Network Architecture), and a dynamically reconfigurable two-dimensional matrix. The DNA is embedded in an array of 376 PE (Processing Elements), which are comprised of computation units, memory, synchronizers, and counters. The PE Matrix circuitry can be reconfigured freely into the structure, which is suitable for application on demand.

It’s very difficult to search in parallel using Dijkstra’s algorithm because of the need for previous calculation results. The proposed algorithm finds the shortest paths in parallel using the PE Matrix. Moreover, we partition the nodes in the network into different groups and merge calculation results after computing the shortest paths using a pipeline operation. Our simulations show that calculation time of the proposed algorithm is less than that of Dijkstra’s algorithm on DAPDNA-2.

The paper is organized as follows. In Section II, we explain Dijkstra’s shortest path algorithm. In Section III, we propose a parallel shortest path algorithm and implement it on DAPDNA-2. Simulation results are provided to evaluate the performance of the proposed algorithm in Section IV. In Section V, we summarize this paper.

II. RELATED WORKS

An OSPF-based network can be divided into smaller networks. A special area called the backbone area forms the core of the network, and other areas are connected to it. Inter-area routing is done via the backbone and all areas must connect to it. If no direct connection is possible, a virtual link may be established[7].

Routers in the same broadcast domain or at either end of a point-to-point telecommunications link form adjacencies when
they have detected each other. This detection occurs when a router sees itself in a hello packet. This is called a two way state and is the most basic relationship. The routers select a designated router (DR) and a backup designated router (BDR), which acts as a hub to reduce traffic between routers. OSPF uses both unicast and multicast to send hello packets and link state updates.

Route calculation is the most important part of OSPF and OSPF routers typically use Dijkstra’s algorithm. This is a simple algorithm that efficiently calculates the shortest paths to all destinations. The single-source shortest path (SSSP) problem is the problem of finding a path between two nodes that minimizes the sum of the weights of its constituent links. The algorithm incrementally organizes a tree structure rooted at the source. All of the router’s neighbors are added to a candidate list, with costs equal to the cost of the links from the router to the neighbors. The router on the candidate list with the smallest cost is then added to the shortest path tree, and that router’s neighbors are then examined to determine whether they can be included in the candidate list. The algorithm then iterates until the candidate list is empty.

Here is an accurate procedure. The input of the algorithm consists of a weighted directed graph \( G = (V, E) \), where \( V \) is a set of nodes and \( E \) is a set of links. Weights of links are given by a weight function \( w : E \rightarrow [0, \infty) \), therefore \( w(u, v) \) is the non-negative cost of moving directly from \( u \) to \( v \) in \( V \). The cost of a link can be thought of as the distance between the two nodes. For each \( v \in V \), \( d(v) \) represents the cost of the shortest path from the source node \( s \) to \( v \).

1) Set \( S \) to empty, where \( S \) is a set of nodes whose shortest paths from the source have already been determined.
2) Add the source node \( s \) to \( S \) and \( d(s) = 0 \). If there is a link from \( s \) to \( v \), \( d(v) = w(s, v) \), for all other nodes, \( d(v) = \infty \).
3) Add a node \( u \) to \( S \), where \( d(u) \) is the smallest in \( V - S \).
   If \( S = V \), complete the task.
4) If there is a link from \( u \) to \( v \in V - S \), \( d(v) = \min\{d(v), d(u) + w(u, v)\} \). Then go back to step 3.

Figure 1 shows an example of Dijkstra’s algorithm. In this example, the network consists of five nodes. Small integers \( w(u, v) \) are assigned to link \((u, v)\), with more preferred links being assigned smaller values than others. Link weights have been assigned symmetrically in this example, although they do not have to be. All nodes have infinite cost except the source. The source node is A and \( d(A) = 0 \). Set \( d(B) = 8 \) because there is a link between A and B as well as \( d(C) = 2 \). Choose the closest node C and add C to S. Update \( d(B) = 5 \), \( d(D) = 11 \), \( d(E) = 6 \) because they are connected with C. Now node B is the closest, add it to S. Update \( d(D) = 6 \), \( d(E) = 6 \). Then node D is the next closest node, choose it and add it to S. Update \( d(E) = 6 \). Finally, add node E to S and the task is completed. Note that only one node can be determined at one time. In other words, the results of the calculations are affected by the next closest node, so it is called the “sequential searching method”.

Dijkstra’s algorithm is commonly used for Internet routing because of its simplicity and suitability for program counter-based processors. The running time of Dijkstra’s algorithm on a graph \( G = (V, E) \) is \( O(V^2) \). The more the number of nodes in a network increases, the more the running time increases. Moreover, it’s very difficult to compute in parallel using Dijkstra’s algorithm because it needs previous calculation results. Dijkstra’s algorithm is strictly limited in its performance even if it uses multi-processor core.

Previous works on dynamic SPT update reduce the computation time compared to the static methods. These works make use of the structure of the previously computed SPT. The objective is to achieve routing stability by making minimum changes to the topology of an existing SPT when some link states in the network have changed. Narvaez et al. established an algorithmic framework to characterize a variety of dynamic SPT algorithms including Dijkstra’s algorithm[8]. Xiao et al. focused on only significantly elements that contribute to the construction of the new SPT from the old one[9]. The efficiency of their algorithms is improved because they only pay attention to the edges really count for the update process. These methods are very efficient when link states have changed slightly. However, these methods can’t improve the shortest path algorithm fundamentally, and are unsuitable for the shortest path calculation based on multiple metrics such as delay or network bandwidth.

### III. PROPOSED SCHEME

#### A. Algorithm summary

The basic idea of our proposal is that a simultaneous multipath search from the source depending on cost determines the shortest paths to each node. The pace of the proceedings changes every time and is equal to the cost of reaching the closest neighbor. A function \( r(u, v) \) represents how far it is from the current position to \( v \) in a link \((u, v)\). The following is an accurate procedure.

1) Set \( S \) to empty. Add the source node \( s \) to \( S \) and \( d(s) = 0 \). For all other nodes \( d(v) = \infty \), and for all links \( r(u, v) = w(u, v) \).
2) Set \(L\) to empty, where \(L\) is a set of links in which current position proceeds. If there is a link from \(s\) to \(v\), add it to \(L\).

3) Take \(r(p, q)\) steps for all \(l \in L\), where \(r(p, q)\) is the minimum value for \((p, q) \in L\). Add \(q\) to \(S\) and \(d(q) = d(p) + w(p, q)\). Set \(r(u, v) = r(u, v) - r(p, q)\) for all \((u, v) \in L\). If \(S = V\), complete the task.

4) Delete \((v, q)\) from \(L\) if there is a link from \(v \in S\) to \(q\). Add \((q, v)\) to \(L\) if there is a link from \(q\) to \(v \in V - S\). Then go back to step 3.

Figure 2 shows an example of our proposed scheme. In this example, the topology is the same as in Figure 1. The denominator represents \(w(u, v)\) and the numerator represents \(w(u, v) - r(u, v)\) in a link \((u, v)\). First, the algorithm adds link AB, AC to \(L\). For these links \(r(A, B) = 8\) and \(r(A, C) = 2\). The minimum value is 2 in this example, so link AC reaches node C. Then it adds C to \(S\) and \(d(C) = 2\). Link AB proceeds two steps. It deletes AC from \(L\) and adds CB, CD, and CE to \(L\). It updates \(r(A, B) = 6\), \(r(C, B) = 3\), \(r(C, D) = 9\), and \(r(C, E) = 4\). The minimum value is 3, so link CB reaches node B. It then adds B to \(S\) and \(d(B) = 5\). Link AB, CD, and CE proceed three steps. It deletes AB and CB from \(L\) and adds BD to \(L\). It updates \(r(B, D) = 1\), \(r(C, D) = 6\), and \(r(C, E) = 1\). The minimum value is one, so link BD reaches node D, and link CE reaches node E simultaneously. It adds D and E to \(S\) and \(d(D) = 0\), \(d(E) = 6\). Link CD proceeds one step. The task is completed because \(S = V\). Note that several nodes can be determined at one time. In other words, in contrast to Dijkstra’s algorithm, the results of these calculations are not affected by the next closest node.

B. Implementation on DAPDNA-2

In this research, we implemented the proposed algorithm on DAPDNA-2 using a matrix-based operation. A network matrix is a matrix representation of network topology where an element represents link weight. A row represents an upstream node, and a column represents a downstream node. When node \(i\) and \(j\) are connected at \(k\) cost, for example, \(a_{ij} = a_{ji} = k\). If node \(i\) and \(j\) are not connected directly, \(a_{ij} = a_{ji} = \infty\). Input data unit is 32bit in PE. 32bit is used as 8bit \(\times\) 4, that is to say, an element is 8bit. The most significant bit is the flag, which means the shortest path is determined and the other 7bits is link weight. When the shortest path to node \(i\) is determined, the flag of 8th line is 1. Otherwise, the flag of 8th line is 0. An 8 \(\times\) 8 network matrix has 64 elements and expresses them in 16 data units because each datum represents four elements.

Figure 3 shows an example of the proposed algorithm using a 5 \(\times\) 5 matrix operation. The source is A. The flag of the first row is 1 and the first column is \(\infty\). First, it evaluates the minimum value of the first row corresponding to the source, which is two in this example, and subtracts it from the first row. This subtraction is a parallel search from the source. Then \(a_{13} = 0\), which means the shortest path to node C is determined. The algorithm then sets the third column to \(\infty\) to prevent node C, corresponding to the third column, from being reduced to 0 by subtraction. It sets the flag of the third row to 1. The flags of the first and the third row are 1 at this moment. Then it evaluates the minimum value from these rows. It iterates calculation in a similar way until all of the flags are 1. When the task is completed, all matrix elements are \(\infty\).

Figure 4 shows an example of the implementation of our algorithm on DAPDNA-2. This module divides 32bit input...
Fig. 4. Implementation example on DAPDNA-2 using parallel and pipeline operation

data into four matrix elements and evaluates the minimum value of each row by tournament. Sequential program requires $N - 1$ comparisons to find the minimum value for $N$ values, while the tournament requires $\log_2 N$ stages. We implemented an $8 \times 8$ matrix operational circuit, so six stages are needed to evaluate the minimum value.

C. The all-node pairs shortest path by pipeline operation

The proposed algorithm uses fewer clock cycles than Dijkstra’s algorithm because it uses the parallel route search method. Our approach can also easily and simultaneously calculate the all-node pair’s shortest path (APSP) search. The required computation time for the APSP search using a pipeline operation is almost the same as for the SSSP search. The data of a pipeline are often executed in parallel and in a time-sliced fashion[6]. Therefore, the data must not depend on each other.

We have to implement an $N \times N$ matrix operational circuit to find the shortest paths of the $N$-node network. Thus, the matrix becomes huge when $N$ is large. To solve this problem, the network is divided into different groups, and calculation results are merged into an abstract network after the shortest respective paths are found.

Figure 5 shows the basic concept of the layered architecture. Small circles represent nodes in this figure, and the bottom layer is given the original network. Edge nodes are the nodes that connect different groups. If a total number of nodes is more than a certain limited value, we partition the network again until the network consists of less than a certain number of nodes. As a result of partitioning, many layers of networks are formed.

Two layers of networks are formed in this figure. First, APSP for all network groups (a, b, c) is calculated simultaneously using a pipeline operation. There are one or more edge nodes in each group. In other words, we can abstract the network by finding APSP. For example, network group b is abstracted by node (b-4, b-6, b-8), and node (b-1, b-2, b-3, b-5, b-7) is eliminated in layer 2. Note that APSP calculations for the network group (a, b, c) in Figure 5 can be done simultaneously. To find the shortest path to a node in group c, for example, the algorithm adds a distance from the source to the edge node in group c (by finding SSSP in layer 2) and a distance from the edge node in group c to the node (by finding APSP in layer 1). Simplification of a network by partitioning enables our algorithm to perform better than Dijkstra’s algorithm.

Figure 6 shows a pipeline operation. For example, a-1 is matrix data which means the source is node a-1 in group a. First, we enter a-1 in a matrix operational circuit, and a-2 after one clock. In a similar fashion, we enter different matrix data for each clock cycle. The pipeline operation is possible because input data from the APSP search are independent of each other. It takes $N$ times as long to find APSP as SSSP on a program counter-based processor when the number of nodes is $N$. However, of APSP’s computation time on DAPDNA-2 increases very little compared to SSSP’s. When we let $c$ be the number of clock cycles required to find SSSP, $c + N - 1$ clocks are required to find APSP on DAPDNA-2, although there are $cN$ clocks on a program counter-based processor. Different matrix data are entered in each clock cycle. While the proposed algorithm is processor dependent, it can be implemented on not only DAPDNA-2 but also other reconfigurable processor.

IV. PERFORMANCE EVALUATION

In this section, we report the results of a simulated comparison of the proposed algorithm and Dijkstra’s algorithm. The parameters in the simulation include the number of nodes $N$, the average number of neighbors $b$, the average number of edge nodes $e$, the longest distance from the source $d$, the number of nodes in a group $m$, and the number of groups $g$. Network topology is formed by connecting nodes with uniform probability. We partition the network into different groups consisting of eight nodes. Edge nodes are given in the proposed algorithm.

It takes $N$ clock cycles to read initial entries, 14 clock cycles to read a neighbor’s data, and 86 clock cycles to execute other
instructions on our circuit of Dijkstra’s algorithm. Therefore, it requires \((N + 14 \times b + 86)(N - 1)\) clock cycles. Figure 7 shows the number of clocks versus \(N\). Let \(b\) be 3 and \(e\) be 2. In the proposed algorithm, clock cycles increase slowly as \(N\) increases because DAPDNA-2 calculates in parallel using a pipeline operation. When \(N = 512\), 99.6% fewer clock cycles are used than with Dijkstra’s algorithm.

When we partition \(N\) nodes into different groups consisting of \(m\) nodes, we think about how many groups might be calculated. For example, when \(N = 24\) and \(m = 8\), we partition the network into three groups, and then merge the calculation results, meaning four groups are calculated in this case. Generally, the numbers of groups that might be calculated are as follows.

\[
g = \left\lfloor \frac{N}{m} \right\rfloor + \left\lfloor \frac{N}{m} \times e \right\rfloor + \left\lfloor \frac{N}{m} \times e \times e \right\rfloor + \cdots + \sum_{i=0}^{j} \left\lfloor \frac{N e^i}{m^{i+1}} \right\rfloor
\]

(1)

where

\[
j = \min \left\{ i \in \mathbb{Z} \mid i \geq 0, \left\lfloor \frac{N e^i}{m^{i+1}} \right\rfloor \leq 1 \right\}
\]

(2)

Given \(x\), the ceiling function denoted by \(\left\lceil x \right\rceil\) is the mathematical function defined as the smallest integer not less than \(x\). For example, \(\left\lceil 2.3 \right\rceil = 3\) and \(\left\lceil 2 \right\rceil = 2\). The first term of the equation (1) is the number of groups in the bottom layer. The second term is the number of groups in the second layer, and the third term is the number of groups in the third layer.

Figure 8 shows the number of clocks versus \(m\). We compare \(m = 8\) with \(m = 16\) and \(m = 32\). It takes 483 clock cycles to solve an \(8 \times 8\) matrix on our circuit. Thus, it takes 483+8g−1
V. Conclusion

In this paper, we have proposed the parallel shortest path-searching algorithm and implemented it in the dynamically reconfigurable processor DAPDNA-2 (IPFlex Inc). Dijkstra’s algorithm, which is the conventional one, finds the shortest paths from the source in OSPF. When the network scale is large, the calculation time for Dijkstra’s shortest path algorithm increases rapidly, so a high-end CPU and memory are required to run it. It’s very difficult to compute in parallel using Dijkstra’s algorithm because it needs previously calculated results. The proposed algorithm finds the shortest paths in parallel using the PE Matrix. Note that, in contrast to Dijkstra’s algorithm, several nodes can be determined at one time. In this research, we implemented the proposed algorithm on DAPDNA-2 using a matrix operation. Moreover, we partitioned the network into small groups and found APSP in each group. The network can be abstracted, and we can easily find the shortest paths from the source. A pipeline operation is possible because the input data of the APSP problem are independent of each other. We have to merge calculation results after finding the shortest respective paths. Simulation results show that the proposed algorithm requires far less computation time than Dijkstra’s algorithm on DAPDNA-2. We show that the number of clocks is independent of $m$ because $mg$ is maintained virtually constant, where $m$ is the number of nodes in a group and $g$ is the number of groups.

Acknowledgment

The authors thank Tomomi Sato and other staff for helping with implementation on DAPDNA-2 (IPFlex Inc). This work was partly supported by the Japan Society for the Promotion of Science’s (JSPS) Grant-in-aid for Scientific Research, by Keio University’s COE Program in “Optical and Electronic Device on Access Network”, by the Japanese Ministry of Education, Culture, Sports, Science, and Technology, and by the Center for Advanced Telecommunication Technology Research, Foundation.

References