FAST PARALLEL ALGORITHMS FOR ROUTING
ONE-TO-ONE ASSIGNMENTS IN BENES NETWORKS

Ching-Yi Lee and A. Yavuz Oruç
Electrical Engineering Department
and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742-3025

Abstract— This paper presents new results on routing unicast (one-to-one) assignments over Beneš networks. Parallel routing algorithms with polylogarithmic routing times have been reported earlier [10, 8], but these algorithms can only route permutation assignments unless unused inputs are assigned to dummy outputs. This restriction is removed in this paper by using techniques that permit bypassing idle or unused inputs without any increase in the order of routing cost or routing time. We realize our routing algorithm on two different topologies. The algorithm routes a unicast assignment involving \(O(k)\) pairs of inputs and outputs in \(O(\log^2 k + \log n)\) time if every pair of processors is interconnected by a direct link, and in \(O(\log^4 k + \log^2 k \log n)\) time if the processors are interconnected by an extended shuffle-exchange network. The same algorithm can be pipelined to route \(\alpha\) unicast assignments, each involving \(O(k)\) pairs of inputs and outputs, in \(O(\log^2 k + \log n + (\alpha - 1) \log k)\) time on the completely connected graph, and in \(O(\log^4 k + \log^2 k \log n + (\alpha - 1)(\log^3 k + \log k \log n))\) time on the extended shuffle-exchange graph. These yield an average routing time of \(O(\log k)\) in the first case, and \(O(\log^3 k + \log k \log n)\) in the second case, for all \(\alpha \geq \log n\). These complexities indicate that the algorithm given in this paper is as fast as the algorithm given in [10] for unicast assignments, and with pipelining it is faster at least by a factor of \(O(\log n)\) on both topologies. Furthermore, for sparse assignments, i.e., when \(k \ll n\), it is the first algorithm which has an average routing time of \(O(\log n)\) on a topology with \(O(n)\) links.

1 INTRODUCTION

The Beneš network has received much attention in interconnection network literature because of its \(O(n \log n)\) cost and \(O(\log n)\) depth [1, 8, 9, 10, 7]. In a way, this network can be considered the forerunner of most multistage interconnection networks that have been extensively studied and used in some real parallel computer systems for interprocessor or processor-memory communications [12, 6].

One problem regarding the Beneš network that remains to have a satisfactory solution is its routing. Many routing algorithms have been reported extending from the \(O(n \log n)\) time looping procedure of Waksman [14], and Opferman and Tsao-Wu [11] to the \(O(\log^2 n)\) time parallel algorithms of Lev et al [8], and Nassimi and Sahni [10]. Other routing algorithms for the Beneš network include matching and edge-coloring schemes [4, 2, 3].

The looping algorithm was developed with the realization that there are two subnetworks in...
the center stage for the network’s inputs to connect to its outputs, and with the added constraint that no two inputs can be connected to two outputs through the same subnetwork in the center stage unless those outputs belong to different switches in the last stage. One gets around this constraint simply by looping between the switches in the first stage and those in the last stage, and assigning the paths in an alternate fashion to the subnetworks in the center stage. Given that there are \( n \) inputs to route, it takes the looping algorithm \( O(n) \) time to set the switches in a Beneš network, and if the same algorithm is applied recursively to the center-stage subnetworks then all the switches in a recursively decomposed Beneš network can be set in \( O(n \log n) \) time.

The routing time of the Beneš network can be reduced in several ways. The most obvious approach is to use a binary tree to set the subnetworks in the center stage in parallel as was done in [3]. The root processor of the tree sets the switches in the first level, its children set the switches in the next level, and so on. It is easy to see that the time complexity of this parallel routing scheme is \( O(n) \). The routing time can be reduced further by introducing parallelism into the setting of switches in each level. This can be done by dividing the outputs into equivalence classes such that two outputs will be in the same class if and only if they must be routed through the same center-stage subnetwork. Once the equivalence classes are decided, the switches in the first and last stages can be set in parallel. The parallel algorithm of Nassimi and Sahni [10] is based on this discovery, and its algorithm takes the parallel computer model and the number of processors available. Their algorithm takes \( O(\log^2 n) \) time on a completely interconnected \( n \)-processor computer, and \( O(\log^4 n) \) time on a shuffle-exchange interconnected \( n \)-processor computer. Lev et al. [8] provided similar parallel algorithms in a more general framework by using edge-coloring schemes. Assuming a parallel computer with conflict free access between \( O(n) \) processors and \( O(n) \) memory elements, their algorithm also takes \( O(\log^2 n) \) time.

While these parallel algorithms are fast, their time complexities are still higher than the \( O(\log n) \) depth of the Beneš network. Furthermore, these algorithms can only route permutation assignments. In case of incomplete assignments where some inputs may remain idle, they cannot be used unless the idle inputs are given dummy outputs. This, however, takes additional time and may render these algorithms inefficient, especially in case of sparse assignments—assignments involving \( O(k) \) pairs of inputs and outputs, where \( k << n \).

In this paper, we present an efficient parallel algorithm for routing incomplete assignments on the Beneš network. We also pipeline this algorithm to attain a factor of \( O(\log n) \) speed up over the parallel algorithms of Nassimi and Sahni and Lev et al. Pipelining is made possible by routing the Beneš network stage by stage from left to right and overlapping the routing steps for consecutive stages. Unlike our routing algorithm, the cited parallel algorithms determine the settings from outer-stage switches toward inner-stage switches. One other difference is that, in our routing scheme, each of the first \( \log n - 1 \) stages is provided with its own special routing module rather than using a single parallel computer to set all switches. Once these stages are set, the stages in the second half are then self-routed as suggested in [7].

Our parallel routing algorithm can run on any parallel computer whose processors are equipped with a constant number of \( O(\log n) \) bit registers and some simple arithmetic and logic circuitry that can compare \( O(\log n) \) bit numbers and perform some counting and decoding functions. We realize our routing schemes on two different topologies. We show that if every pair of processors is interconnected by a direct arc then routing an assignment involving \( O(k) \) pairs of inputs and outputs takes \( O(\log^2 k + \log n) \) time without pipelining and \( O(\log k) \) time with pipelining. We also establish that using a weaker topology, namely the extended shuffle-exchange graph (which will be defined in Section 4), leads to a routing algorithm with \( O(\log^4 k + \log^2 k \log n) \) time without pipelining and \( O(\log^3 k + \log k \log n) \) time with pipelining.
2 BASIC FACTS AND DEFINITIONS

An $n$-network is a directed acyclic graph with $n$ distinguished source vertices, called inputs, $n$ distinguished sink vertices, called outputs, and some internal vertices, called switches. An assignment for an $n$-network is a pairing of its inputs with its outputs such that each output appears in at most one pair. An assignment consisting of $k$ pairs will be called a $k$-assignment. An assignment is called one-to-one or unicast if each input appears in at most one pair. A permutation assignment for an $n$-network is a unicast $n$-assignment. An $n$-network is said to realize an assignment if, for each pair $(a, b)$ in the assignment, a path can be formed from input $a$ to output $b$ by setting the switching nodes in the network with the constraint that the paths for no two pairs $(a, b)$ and $(c, d)$ overlap unless $a = c$. An $n$-network that can realize all unicast assignments is called a unicast $n$-network.

The well-known Beneš network is a unicast $n$-network that is constructed recursively as shown in Figure 1. Each of the first and the last stages consists of $n/2 \times 2 \times 2$ switches, and the center stage consists of two $n/2$-input Beneš networks. Each $2 \times 2$ switch can be set in two ways: either through state where the two inputs are connected straight-through to the two outputs, or cross state where the two inputs are connected to opposite outputs. The inputs and outputs of the upper $n/2$-input Beneš network are numbered $0, 2, \ldots, n - 2$, and the inputs and outputs of the lower $n/2$-input Beneš network are numbered $1, 3, \ldots, n - 1$, from top to bottom. If the half-size Beneš networks in the center stage are recursively decomposed then one obtains a $(2 \lg n - 1)$-stage network consisting of $2 \times 2$ switches, assuming that $n$ is a power of 2.

Paths in an $n$-network will be established by specifying some routing information at its inputs. It is assumed that each input holds its own routing information unless otherwise stated. It is also assumed that the routing information for each input is accompanied by some binary coded message that is to be routed from that input to the output specified in the routing information. A message and routing information combined together will be termed a packet. For an $n$-input Beneš network, the routing part of a packet, to be called the header, is assumed to have $\lg n + 1$ bits, and will be denoted as $(r_i, d^i_{\lg n - 1}, \ldots, d^i_1, d^i_0)$ for a packet at input $i$. The bit $r_i$ specifies whether input $i$ is paired with some output. Input $i$ is said to be busy if $r_i = 1$, and it is said to be idle if $r_i = 0$. The remaining bits, i.e., $(d^i_{\lg n - 1}, \ldots, d^i_1, d^i_0)$, form the output address which specifies the binary representation of the output paired with input $i$ with $d^i_{\lg n - 1}$ being the most significant bit. Besides, a switch is said to be busy if both of its inputs are busy; it is said to be semi-busy if one of its inputs is busy and the other input is idle, and it is said to be idle if both of its inputs are idle.

In light of these facts, we make the notion of a unicast assignment more precise.

Definition 1: A unicast $k$-assignment for an $n$-network is a set $\{(i, (r_i, d^i_{\lg n - 1}, \ldots, d^i_1, d^i_0)) : 0 \leq i \leq n - 1\}$ such that exactly $k$ $r_i$’s are equal to 1, and $(d^i_{\lg n - 1}, \ldots, d^i_1, d^i_0) \neq (d^j_{\lg n - 1}, \ldots, d^j_1, d^j_0)$ whenever $i \neq j$ and $r_i = r_j = 1$. 

3 THE ROUTING PRINCIPLE

In this section, we describe a routing principle which establishes that unicast assignments for the Beneš network can be recursively decomposed into half-sized unicast assignments stage by stage from left to right.

Notation: For $0 \leq i \leq n - 1$, if $(b_{\lg n - 1}, \ldots, b_i, b_0)$ is the binary representation of $i$, then $\tilde{i}$ denotes the integer which has the
binary representation \((b_{i_{g-n-1}}, \ldots, b_1, \overline{b}_0)\), and \(i\) and \(j\) are called a dual pair of integers. ||

Now let \(H(n)\) denote a single stage \(n\)-network which comprises \(n/2 \times 2 \times 2\) switches, \(SW_0, SW_1, \ldots, SW_{n/2-1}\).

**Definition 2:** Given a unicast assignment \(\{(i, (r_i, d^i_{l_{g-n-1}}, \ldots, d^i_1, d^i_0)) : 0 \leq i \leq n-1\}\) for \(H(n)\), a sequence of switches \(SW_{i_0}, SW_{i_1}, \ldots, SW_{i_{p-1}}\) in \(H(n)\) is said to form a chain with respect to that assignment if, for all \(0 \leq q \leq p-2\), one of the inputs of \(SW_{i_q}\) and one of the inputs of \(SW_{i_{q+1}}\) are paired with a dual pair of outputs and have their connecting bits set to 1, i.e., there exist \(x = 2i_q\) or \(x = 2i_q + 1\) and \(y = 2i_{q+1}\) or \(2i_{q+1} + 1\) for which \((d^x_{l_{g-n-1}}, d^x_2, d^x_1, d^x_0) = (d^y_{l_{g-n-1}}, d^y_2, d^y_1, d^y_0)\) and \(r_x = r_y = 1\). Furthermore, \(SW_{i_0}, SW_{i_1}, \ldots, SW_{i_{p-1}}\) is said to be a closed chain if the other input of \(SW_{i_{p-1}}\) and the other input of \(SW_{i_0}\) are also paired with a dual pair of outputs. It is said to be an open chain otherwise. ||

The size of a chain is the number of switches in the chain. Given a unicast \(k\)-assignment for \(H(n)\), the size of a chain can be as large as \(\min\{(k+2)/2, n/2\}\) and as small as 1, and the number of chains can be as large as \(\min\{k, n/2\}\) (when each chain has size 1) and as small as 1 (when that chain is of size \((k+2)/2\) or \([k/2]\)). Figure 2 shows a closed chain \(C_1\) and an open chain \(C_2\) with respect to a unicast 15-assignment for \(H(n)\) where \(n = 16\).

An open chain, say \(SW_{i_0}, SW_{i_1}, \ldots, SW_{i_{p-1}}\), has two end switches (i.e., \(SW_{i_0}\) and \(SW_{i_{p-1}}\)). Based on the status of the end switches, two types of open chains are distinguished.

**Definition 3:** An open chain is said to be a full open chain if both of its end switches are busy or semi-busy, and it is said to be a half open chain if one of its end switches is busy and the other end switch is semi-busy. ||

For example, \(C_2\) given in Figure 2 is a half open chain since \(SW_5\) is busy and \(SW_3\) is semi-busy.

**Theorem 1:** Given \(\{(i, (r_i, d^i_{l_{g-n-1}}, \ldots, d^i_1, d^i_0)) : 0 \leq i \leq n-1\}\), a unicast \(k\)-assignment for \(H(n)\), let \((r'_i, p^i_{l_{g-n-1}}, \ldots, p^i_1, p^i_0)\) denote the header of the packet at output \(i\) of \(H(n)\), \(0 \leq i \leq n-1\). There exist settings for \(SW_0, SW_1, \ldots, SW_{n/2-1}\) such that \(\{(2i, (r'_{2i}, p^i_{l_{g-n-1}}, \ldots, p^i_1, p^i_2)) : 0 \leq i \leq n/2-1\}\) is a unicast \(k_0\)-assignment and \(\{(2i + 1, (r'_{2i+1}, p^i_{l_{g-n-1}}, \ldots, p^i_2, p^i_1)) : 0 \leq i \leq n/2-1\}\) is a unicast \(k_1\)-assignment, where \(k_0 = [k/2]\) and \(k_1 = [k/2]\).

**Proof:** With respect to the given unicast \(k\)-assignment and without loss of generality, suppose that there exist \(c\) chains, \(1 \leq c \leq \min\{k, n/2\}\), and \(f\) of these \(c\) chains are full open chains, \(0 \leq f \leq c\). From Definition 2, the switches of each chain can be set such that, given inputs \(i\) and \(j\) which have \(r_i = r_j = 1\) and \((d^i_{l_{g-n-1}}, \ldots, d^i_2, d^i_1, d^i_0) = (d^j_{l_{g-n-1}}, \ldots, d^j_2, d^j_1, d^j_0)\), one is routed to an even-numbered output and the other is routed to an odd-numbered output, and once a switch of the chain is set then the settings of the other switches of the chain are fixed. Moreover, different chains can be set mutually independently resulting in \(2^c\) settings since each chain has two ways of settings. It is obvious that, with each of these \(2^c\) settings in \(H(n)\), \(\{(2i, (r'_{2i}, p^i_{l_{g-n-1}}, \ldots, p^i_2, p^i_1)) : 0 \leq i \leq n/2-1\}\) is a unicast \(k_0\)-assignment and \(\{(2i + 1, (r'_{2i+1}, p^i_{l_{g-n-1}}, \ldots, p^i_2, p^i_1)) : 0 \leq i \leq n/2-1\}\) is a unicast \(k_1\)-assignment for some integers \(k_0\) and \(k_1\) with \(k_0 + k_1 = k\). Thus,

---

2\(b\) denotes the binary complement of bit \(b\).
it suffices to show at least one of the $2^c$ settings will result in $k_0 = \lfloor k/2 \rfloor$ and $k_1 = \lceil k/2 \rceil$. The closed and full open chains, no matter how they are set, will increase $k_0$ and $k_1$ by the same integer since such chains have even numbers of busy inputs. However, in one setting of a half open chain (Type-0 Setting), $k_0$ will increase one more than $k_1$, and in its other setting (Type-1 Setting), $k_0$ will increase one less than $k_1$. Suppose that $\lceil f/2 \rceil$ of the $f$ full open chains are in their Type-0 Settings and the other $\lfloor f/2 \rfloor$ full open chains are in their Type-1 Settings. Then, $k_0 = \lfloor k/2 \rfloor$ and $k_1 = \lceil k/2 \rceil$, and the statement follows. ||

Given a unicast assignment for an $n$-input Beneš network, let the switches in the first stage be set such that the statement of Theorem 1 is satisfied. Then the two established half-sized assignments for the two center-stage $n/2$-input Beneš networks are also unicast and can be realized by these networks, respectively. Hence, any algorithm that satisfies the statement of Theorem 1 can be used recursively to set the switches in the first $\lg n - 1$ stages. Thereafter the packets can be routed on a self-routing basis through the last $\lg n$ stages to their final destinations as identified before [7].

4 THE PARALLEL ROUTING ALGORITHM

Following the discussion in the previous section, it is only necessary to describe a parallel algorithm for $H(n)$ so that the routing principle stated in Theorem 1 is satisfied. The parallel routing algorithm for the Beneš network then follows.

In the parallel algorithm for $H(n)$, it is assumed that there are $n$ interconnected processors, $PR(0), PR(1), \ldots, PR(n-1)$, and that a packet header $(r_i, d_{i0}, \ldots, d_{in-1})$ is initially input to $PR(i)$. $PR(i)$ and $PR(i)$ are called a dual pair of processors and will determine the setting of $SW_{i/2}$ of $H(n)$, $0 \leq i \leq n - 1$. The time complexity of the parallel algorithm depends on the interconnection topology between these $n$ processors. The algorithm will run on two connection topologies, namely the completely connected graph and the extended shuffle-exchange network.

A. THE ROUTING MODEL

The parallel algorithm uses three kinds of macro functions: move process, concentrate process and broadcast process for exchanging packets between the $n$ processors. A move process transfers packets from a subset of processors to a subset of processors in a one-to-one manner. A concentrate process transfers packets from a subset of processors to a single processor in a many-to-one manner. A broadcast process transfers packets from a single processor to a subset of processors in a one-to-many manner.

The time complexity to execute any of these processes depends on the topology which interconnects the processors. We will consider two topologies. The first is the completely connected graph in which there is a link between every two processors. It is obvious that each of the above three processes can be executed in $O(1)$ time over the completely connected graph.

The second topology we will use is called the extended shuffle-exchange network in which (a) each dual pair of processors is connected by a link, and (b) processors $PR(0), PR(1), \ldots, PR(n/2^m - 1)$ are $(n/2^m)$-shuffle connected for $m = 0, 1, \ldots, \lg n - 2$, (i.e., $PR(0), PR(1), \ldots, PR(n-1)$ is $n$-shuffle connected, $PR(0), PR(1), \ldots, PR(n/2 - 1)$ is $n/2$-shuffle connected, and so on), where the replicated links between the processors due to the shuffle connections are coalesced together. Essentially, the extended shuffle-exchange network is obtained by superposing Stone’s perfect shuffle-exchange network [13,5] for $n, n/2, \ldots, 4$ inputs. Compared with the $O(n^2)$ links needed for the completely connected graph, it is easily verified that the extended shuffle-exchange network requires $O(n)$ links which is also the case with Stone’s perfect shuffle-exchange network.

Any concentrate or broadcast process involving any subset of the processors can be completed over an extended shuffle-exchange network by passing $O(\lg n)$ times through the $n$-shuffle links between $PR(0), PR(1), \ldots, PR(n-1)$, and hence each of the concentrate and broadcast processes can be executed in $O(\lg n)$ time. To execute a move process, the extended shuffle-exchange graph takes three steps. Suppose that some
$k$ processors have packets (one each) which are to be moved to some $k$ processors, $2 \leq k \leq n$ ($k = 1$ is a trivial case). In the first step, the $k$ packets are moved to any $k$ of the first $2^{\lceil \lg k \rceil}$ processors by passing $\lg n$ times through the $n$-shuffle connection. In the second step, these $k$ packets are sorted according to their destination addresses by passing $\lg^2 k'$ times through the $k'$-shuffle interconnection between $PR(0), PR(1), \ldots, PR(k' - 1)$, where $k' = 2^{\lceil \lg k \rceil}$ [13, 5]. In the third step, the sorted packets are moved to their final destinations by passing them $\lg n$ times through the $n$-shuffle connection. Therefore, a move process involving $k$ packets can be executed over the extended shuffle-exchange graph in $O(\lg^2 k + \lg n)$ time. We note that, the extended shuffle-exchange network was intentionally designed to contain a shuffle exchange network for each power of 2 number of inputs between 1 and $n$. This is needed since the value of $k'$ varies between 1 and $n$.

**B. The Parallel Routing Scheme**

From the proof of Theorem 1 in Section 3, each chain has two settings (Type-0 and Type-1 Settings), and the inputs of its switches can be partitioned into two equivalence classes such that the inputs in one equivalence class are connected to even-numbered outputs and the inputs in the other equivalence class are connected to odd-numbered outputs. Once the equivalence classes of inputs in a chain are established, the settings of switches in the chain are straightforward. The parallel algorithm that follows will use such equivalence classes to speed up the switch settings. The following proposition shows how to determine the pairs of inputs that are in the same equivalence class, and is similar to the observation given on p. 150 in [10].

**Proposition:** Let 
$\{(i, (r_i, d_{\lg n - 1}^i, \ldots, d_1^i, d_0^i)): 0 \leq i \leq n - 1\}$ be a unicast assignment for $H(n)$. If $(d_{\lg n - 1}^i, \ldots, d_1^i, d_0^i) = (d_{\lg n - 1}^j, \ldots, d_1^j, d_0^j)$, $r_i = r_j = 1$ and $i \neq j$, then inputs $i$ and $j$ are in the same equivalence class and inputs $j$ and $\tilde{i}$ are in another equivalence class.

**Proof:** The proof is straightforward and is omitted. ||

**Remark:** This equivalence among the inputs will be noted by associating each input with an ordered quadruple in which the first element is a single bit used to indicate if this input belongs to a closed or open chain, the second element corresponds to this input, the third element points to an input that is in the same equivalence class as this input, and the fourth element, called the representative of this input, will be used to route this input. If $i$ and $j$ satisfy the hypothesis of Proposition 1, two ordered quadruples $(1; i, j; p_i)$ and $(1; j, \tilde{i}; p_j)$ will be established, where their first elements are initialized to 1 and their fourth elements are initialized to $p_i = \text{min}\{i, j\}$ and $p_j = \text{min}\{j, \tilde{i}\}$. If $r_i = 1$ and $r_j = 1$ for which $(d_{\lg n - 1}^i, \ldots, d_1^i, d_0^i) = (d_{\lg n - 1}^j, \ldots, d_1^j, d_0^j)$ (i.e., input $i$ is paired with an output whose dual output is idle), an ordered quadruple $(1; i; -1; \tilde{i})$ will be established, where $-1$ is used to denote that this input belongs to a busy end switch of an open chain, or an open chain of size 1. If $r_i = 0$ and $r_j = 1$, an ordered quadruple $(0; i, -2; \tilde{i})$ will be established, where $-2$ is used to denote that this input belongs to a semi-busy end switch of an open chain. ||

The parallel algorithm can be roughly divided into four phases. In the first phase, ordered quadruples as defined in Remark 1 are established. In the second phase, the representative in each quadruple is computed such that each input knows to which chain it belongs and all the inputs in the same equivalence class will agree on a common representative. In the third phase, each half open chain is assigned a Type-0 or Type-1 Setting so that the statement of Theorem 1 (i.e., $k_0 = \lceil k/2 \rceil$ and $k_1 = \lfloor k/2 \rfloor$) is satisfied. In the fourth phase, each switch is set by using the representatives of its two inputs.

Using the unicast assignment given in Figure 2 as an example, the operation of each phase is outlined as follows. The first phase applies Remark 1 to establish ordered quadruples, and can be further decomposed into three steps. At the first step, packet headers are input to the processors, and the idle inputs which belong to semi-busy switches have their quadruples established. At the second step, packet headers are moved to new processors so that each dual pair of processors can apply the steps in Remark 1.
in parallel. At the third step, the remaining quadruples are established and moved to new processors which are specified by their second elements, and their first elements are changed to 0 if their third elements are -1. An illustration of these three steps is shown in Figure 3, where a column with 16 entries is used to express the data stored in the 16 processors at each step.

The second phase is an iterative procedure which computes the representative of each quadruple. This computation is the crux of the parallel algorithm, and it needs to be explained in detail.

A chain, depending upon its type, is decomposed into two sequences of quadruples in the first phase where the quadruples in each sequence belong to the same equivalence class. Let us use the quadruples in each sequence as nodes to form a directed graph in which a directed arc is established from a quadruple to another quadruple if and only if the second element of the former quadruple is equal to the second element of the latter quadruple. Then, a $k$-size closed chain will form two $k$-node closed subchains for which each node has an incoming arc and an outgoing arc, and a $k$-size open chain will form two $k$-node open subchains for which each source node has an outgoing arc, each sink node has an incoming arc, and each of the other nodes has an incoming arc and an outgoing arc. For example, Figure 4 explicitly depicts the four subchains obtained from the quadruples in Figure 3(c).

These subchains will be used to facilitate an understanding of the iterative procedure. If a quadruple is in a closed subchain in the $m$-th iteration, its first element is 1, its third element points to the second element of its $2m$-th successor and its last element points to the smallest input among the $2m$ second elements of its first $2m$ successors. If a quadruple is in an open subchain, the way its elements are updated depends on when it is known that this quadruple is in an open subchain. When this quadruple is at distance less than $2^m - 1$ away from the sink quadruple in the $m$-th iteration, its elements are updated in the same way as if this quadruple is in a closed subchain. When this quadruple is at distance less than $2^m - 1$ away from the sink quadruple in the $m$-th iteration, it is recognized to be in an open subchain, and its first element is changed to 0, its third element is changed to -1 or -2 (the same as the third element of the sink quadruple) and its last element points to the second element of the sink quadruple. That is, for each open subchain, the updating information is exponentially propagated from the sink quadruple to the source quadruple. Using this updating procedure, after $\log[k]$ iterations, any $k$ quadruples that form a closed subchain will select the smallest input among their second elements as their common representative, and any $k$ quadruples that form an open subchain will select the second element of the sink quadruple as their common representative. For example, the four subchains in Figure 4 need two iterations to determine their representatives, and inputs 2, 3, 11 and 6 will be selected as the representatives of the quadruples in $SC_1'$, $SC_1''$, $SC_2'$ and $SC_2''$, respectively.

Figure 3: The three steps of the first phase by using the unicast 15-assignment given in Figure 2 as an example.

Figure 4: The subchains formed from the quadruples in Figure 3(c).
Now we proceed to describe the second phase of the algorithm. Given a unicast \(k^2\)-assignment, the second phase is assumed to consist of \(\lceil \log k \rceil \) iterations, where \(k = \min\left\{(k^2 + 2)/2, n/2\right\}\). This assumption is justified since there is no prior information about the exact sizes of the subchains and \(\min\left\{(k^2 + 2)/2, n/2\right\}\) is an upper bound for the sizes of the subchains with respect to the \(k^2\)-assignment as stated in the previous section. Technically, each iteration can be further decomposed into three steps.

In the second step, each processor updates the quadruple(s) that it holds as follows: (1) when it holds only a quadruple, it “kills” the quadruple if the first element of the quadruple is 1; otherwise it keeps the quadruple intact; (2) when it holds \(\langle 1; l, i; p_i \rangle\) and \(\langle 1; i; j; p_i \rangle\), it replaces the first quadruple by \(\langle 1; l, j; p_i \rangle\) where \(p'_i = \min\{p_i, p_l\}\) and kills the second quadruple; (3) when it holds \(\langle 1; l, i; p_i \rangle\) and \(\langle 0; i; *; j \rangle\) where \(*\) is \(-2\) or \(-1\), it replaces the first quadruple by \(\langle 1; l, *; i \rangle\) and keeps the second quadruple intact.

In the third step, each updated quadruple is moved to a new processor specified by its second element if its first element remains 1, and its copies are moved to new processors specified by their third elements. In the second step, each processor updates its quadruple(s) that it holds as follows: (1) when it holds only a quadruple, it “kills” the quadruple if the first element of the quadruple is 1; otherwise it keeps the quadruple intact; (2) when it holds \(\langle 1; l, i; p_i \rangle\) and \(\langle 1; i; j; p_i \rangle\), it replaces the first quadruple by \(\langle 1; l, j; p_i \rangle\) where \(p'_i = \min\{p_i, p_l\}\) and kills the second quadruple; (3) when it holds \(\langle 1; l, i; p_i \rangle\) and \(\langle 0; i; *; j \rangle\) where \(*\) is \(-2\) or \(-1\), it replaces the first quadruple by \(\langle 1; l, *; i \rangle\) and keeps the second quadruple intact.

In the third step, each updated quadruple is moved to a new processor specified by its second element if its first element remains 1, and its copies are moved to new processors specified by their third elements.

For example, we illustrate the second phase in (e) of Figure 5, where the transitions from (a) to (b) and from (b) to (c) constitute the first iteration, and the transitions from (c) to (d) and from (d) to (e) constitute the second iteration. Note that, after the computation of the second phase, the two quadruples held by \(PR(i)\) and \(PR(\tilde{i})\) correspond to inputs \(i\) and \(\tilde{i}\) of \(SW_{i/2}\). Besides, if the first elements of the two quadruples are 1, \(SW_{i/2}\) belongs to a closed chain; otherwise, it belongs to an open chain. Furthermore, \(SW_{i/2}\) belongs to a full open chain if the third elements are both \(-1\) or \(-2\), and to a half open chain if one of the third elements is \(-1\) and the other is \(-2\).

The third phase assigns types of settings to half open chains as discussed in the proof of Theorem 1, and can be decomposed into three parts. In the first part, for each half open chain, the representative of the quadruple that corresponds to the busy input of the semi-busy end switch is concentrated to a specific processor, say \(PR(0)\). In the second part, \(PR(0)\) assigns Type-0 Setting to half of the concentrated representatives and Type-I Setting to the other half, and broadcasts this information to all the processors. In the third part, each half open chain has its quadruples determine the type of settings it is assigned. For example, Figure 5(f) shows a result of the third phase, where the quadruples in \(PR(0), PR(7), PR(11)\) and \(PR(12)\) have their third elements change to 0 since these quadruples belong to half open chain \(C_2\) which is assigned the Type-0 Setting.

Having decided the types of each chain and computed the representative of each quadruple, the fourth phase determines the settings of switches in \(H(n)\). Each switch is set by a dual pair of processors which hold the two quadruples that correspond to the inputs of
this switch, and the setting depends on the type of the chain to which this switch belongs.

**Case 1:** If \( PR(i) \) holds \( \langle 1; i, k; j \rangle \) and \( PR(i) \) holds \( \langle i; l, j \rangle \), then \( SW_{[1/2]} \) is in a closed chain. Assuming that \( SW_{[j/2]} \) is set through, \( SW_{[i/2]} \) must be set through if \( i - j \) is even and cross otherwise.

**Case 2:** If \( PR(i) \) holds \( \langle 0; i, p; j \rangle \) and \( PR(i) \) holds \( \langle 0; i, -2; l \rangle \) where \( p = 0 \) or \( p = -2 \), then \( SW_{[i/2]} \) is in a full open chain, and \( SW_{[j/2]} \) and \( SW_{[1/2]} \) are the end switches. Assuming that the smaller one of \( SW_{[j/2]} \) and \( SW_{[i/2]} \) is set through, when \( j < l, SW_{[i/2]} \) must be set through if \( i - j \) is even and cross otherwise, and when \( l < j, SW_{[i/2]} \) must be through if \( i - 1 \) is even and cross otherwise.

**Case 3:** If \( PR(i) \) holds \( \langle 0; i, q; j \rangle \) and \( PR(i) \) holds \( \langle 0; i, -2; l \rangle \) where \( q = 0 \) or \( q = 1 \), then \( SW_{[i/2]} \) is in a half open chain and \( SW_{[j/2]} \) is the semi-busy end switch. When \( q = 0 \) (the half open chain is assigned Type-0 Setting), \( SW_{[i/2]} \) must be set through if \( i \) is even and cross otherwise, and when \( q = 1 \) (the half open chain is assigned Type-1 Setting), \( SW_{[i/2]} \) must be set through if \( i \) is odd and cross otherwise.

For example, switches \( SW_0, SW_1, SW_4, SW_6 \) and \( SW_7 \) in Figure 2 are set through and switches \( SW_2, SW_3 \) and \( SW_5 \) are set cross by checking the final quadruples held in each dual pair of processors as shown in Figure 5(f).

**C. The Parallel Algorithm**

The following parallel routing algorithm formalizes the steps outlined in the previous subsection.

**Step 1:** Given \( (i, (r, d_{[\lg n - 1]}, \ldots, d_1, d_0)) \) input to \( PR(i) \), \( PR(i) \) establishes \( \langle 0; i, -2; j \rangle \) if \( r_i = 0, 0 \leq i \leq n - 1 \). Let \( k' \) be the number of \( r_i \)'s whose value are 1, and let \( k = \min \{[(k' + 2)/2], n/2\} \). Let \( m \) be a parameter initialized to 0.

**Step 2:** Move \( (i, (r, d_{[\lg n - 1]}, \ldots, d_1, d_0)) \) to \( PR_x \) if \( r_i = 1 \) and \( (d_{[\lg n - 1]}, \ldots, d_1, d_0) \) is the binary representation of \( x \), \( 0 \leq i \leq n - 1 \).

**Step 3:** Given \( PR_x \) holding \( (i, r_i) \) and \( PR(\bar{x}) \) holding \( (j, r_j) \), \( PR(x) \) establishes \( \langle 1; i, j; p_i \rangle \) and \( PR(\bar{x}) \) establishes \( \langle 1; j, i; p_j \rangle \) where \( p_i = \min \{i, j\} \) and \( p_j = \min \{j, i\} \) if \( r_i = r_j = 1 \), or \( PR(x) \) establishes \( \langle 1; i, -1; i \rangle \) if \( r_i = 1 \) and \( r_j = 0 \), or \( PR(\bar{x}) \) establishes \( \langle 1; j, -1; j \rangle \) if \( r_i = 0 \) and \( r_j = 1 \), \( 0 \leq x \leq n - 1 \). Then, Move \( \langle 1; i, j; p_i \rangle \) to \( PR(i) \), and if the third element is “-1” then \( PR(i) \) changes the first element to 0, \( 0 \leq i \leq n - 1 \).

(Step 1, Step 2 and Step 3 constitute the first phase.)

**Step 4:** \( m = m + 1 \). If \( m \leq \lceil \lg k \rceil \) then go to Step 5, else go to Step 8.

**Step 5:** Duplicate \( \langle 1; l, i, p_i \rangle \) and move a copy to \( PR(i) \), \( 0 \leq i \leq n - 1 \). (Those quadruples whose first elements are 1 are duplicated and moved to new processors specified by their third elements.)

**Step 6:** (The action of each processor depends on the quadruple(s) that it holds.)

(1) when \( PR(i) \) holds only a quadruple: If the first element of the quadruple is 1 then \( PR(i) \) kills the quadruple, else \( PR(i) \) keeps the quadruple intact;

(2) when \( PR(i) \) holds \( \langle 1; l, i, p_i \rangle \) and \( \langle 1; i, j; p_i \rangle : PR(i) \) replaces the first quadruple by \( \langle 1; l, j, p_i' \rangle \) where \( p_i' = \min \{p_i, p_i\} \) and kills the second quadruple;

(3) when \( PR(i) \) holds \( \langle 1; l, i, p_i \rangle \) and \( \langle 0; i, *, j \rangle \) where \( * = -2 \) or \( -2 : PR(i) \) replaces the first quadruple by \( \langle 1; l, *, i \rangle \) and keeps the second quadruple intact.

**Step 7:** Move \( \langle 1; i, j; p_i \rangle \) to \( PR(i) \), and if the third element is “-1” or “-2” then \( PR(i) \) changes the first element to 0, \( 0 \leq i \leq n - 1 \). Go to Step 4.

(Step 4, Step 5, Step 6 and Step 7 constitute the second phase.)

**Step 8:** If \( PR(i) \) holds \( \langle 0; i, -1; j \rangle \) and \( PR(\bar{i}) \) holds \( \langle 0; \bar{i}, -2; i \rangle \) (i.e., input \( i \) is the busy input of \( SW_{[i/2]} \) which is the semi-busy switch of a half open chain), then representative \( j \) of input \( i \) is concentrated \( PR(0), 0 \leq i \leq n - 1 \). Upon receiving the concentrated numbers, say \( j_1, j_2, \ldots, j_f \), \( PR(0) \) assigns 0 to the first \( \lceil f/2 \rceil \) numbers and 1 to the last \( \lceil f/2 \rceil \) numbers, and broadcasts the sequence, \( (j_1, j_2, \ldots, j_f) \), to \( PR(i) \), \( 0 \leq i \leq n - 1 \). Then, the third element of the quadruple held in \( PR(i) \) is changed from -1 to 0 if the fourth element is among the first \( \lceil f/2 \rceil \) numbers of the broadcast sequence, and from -1 to 1 if the fourth element is among the last \( \lceil f/2 \rceil \) numbers of the broadcast sequence, \( 0 \leq i \leq n - 1 \).
Step 9: Case 1: If \( PR(i) \) holds \((1; i, j; p_i)\) and \( PR(\bar{i}) \) holds \((1; \bar{i}, \bar{j}; \bar{p}_i)\), then \( PR(i) \) sets \( SW_{\lfloor i/2 \rfloor} \) through if \( i - p_i \) is even and cross otherwise, \( 0 \leq i \leq n - 1 \).

Case 2: If \( PR(i) \) holds \((0; i, p; j)\) and \( PR(\bar{i}) \) holds \((0; \bar{i}, \bar{p}; \bar{j})\) where \( p = -1 \) or \( p = -2 \), then they compare \( j \) and \( l \). When \( j < l \), \( PR(i) \) sets \( SW_{\lfloor i/2 \rfloor} \) through if \( i - j \) is even and cross otherwise, and when \( l < j \), \( PR(\bar{i}) \) sets \( SW_{\lfloor \bar{i}/2 \rfloor} \) through if \( \bar{i} - l \) is even and cross otherwise, \( 0 \leq i \leq n - 1 \).

Case 3: If \( PR(i) \) holds \((0; i, q; j)\) and \( PR(\bar{i}) \) holds \((0; \bar{i}, -2; \bar{l})\) where \( q = 0 \) or \( q = 1 \), \( PR(i) \) sets \( SW_{\lfloor i/2 \rfloor} \) through if \( i - q \) is even and cross otherwise, \( 0 \leq i \leq n - 1 \).

(Step 9 constitutes the fourth phase.)

That this algorithm is correct can easily be proved and is omitted for lack of space. In the algorithm, move processes dominate the time complexity, and each move process can be executed in \( O(1) \) time if the processors are completely interconnected, and in \( O(\log^2 k + \log n) \) time if the processors are extended shuffle-exchange interconnected, as shown in Subsection A. For a unicast \( k \)-assignment, since there are \( O(\log k) \) move processes in the algorithm, the switches in \( H(n) \) can be set in \( O(\log k) \) time if the interprocessor topology is the complete graph, and in \( O(\log^3 k + \log k \log n) \) time if the interprocessor topology is extended shuffle-exchange graph. Furthermore, as discussed in the previous section, this algorithm can be recursively applied to set switches in the first half stages. In fact, only the first \( \lfloor \log k \rfloor \) stages would apply this algorithm for a unicast \( k \)-assignment since this algorithm decomposes an assignment into two half-sized assignments stage by stage as required in the statement of Theorem 1. Therefore, by using this parallel algorithm, the routing time for an \( n \)-input Beneš network to realize a unicast \( k \)-assignment becomes \( O(\log^2 k + \log n) \) if the interprocessor connection topology is complete, and becomes \( O(\log^4 k + \log^2 k \log n) \) if the interprocessor connection topology is the extended shuffle-exchange network.

Also, unicast assignments can be pipelined over the stages of the Beneš network by using this parallel routing algorithm. That is, when

the switch settings in a stage for a unicast assignment are finished, the switch settings in that stage for another unicast assignment can proceed. Pipelining will reduce the average routing time needed to realize a series of unicast assignments. Suppose that there are \( \alpha \) consecutive unicast assignments to be realized over an \( n \)-input Beneš network. Without pipelining, the average routing time to realize an assignment is \( O(\log^2 k + \log n) \) if the interprocessor connection topology is complete, and is \( O(\log^4 k + \log^2 k \log n) \) if the interprocessor connection topology is the extended shuffle-exchange network. With pipelining, the total routing time to realize these \( \alpha \) assignments reduces to \( O(\log^2 k + \log n + (\alpha - 1) \log k) \) for completely connected network, and to \( O(\log^4 k + \log^2 k \log n + (\alpha - 1)(\log^3 k + \log k \log n)) \) for extended shuffle-exchange network. The average routing time to realize an assignment is reduced to \( O(\log k) \) in the first case, and to \( O(\log^3 k + \log k \log n) \) in the second case, for \( \alpha \geq \log n \).

5 CONCLUDING REMARKS

The paper presented a nontrivial extension of Nassimi and Sahni’s parallel algorithm for routing unicast assignments in Beneš networks. Unlike the original algorithm of Nassimi and Sahni, the new algorithm can route all unicast assignments (including the permutation assignments) without using dummy outputs. In general, on a parallel processor with a completely connected or extended shuffle-exchange network, the time complexity of this algorithm increases polylogarithmically with the size of the assignments, and it matches the time complexity of Nassimi and Sahni’s algorithm for permutation assignments.

References

[1] V. Beneš. Mathematical Theory of Con-


