Some Parallel Algorithms Based on Parentheses Matching on Linear Arrays with Optical Buses

Young-Hak Kim

Kumoh National Institute of Technology, Gumi, Korea

Summary
The parentheses matching problem is to determine the index of the mate for each parenthesis, and plays an important role in the design of parallel algorithms. In this paper, we consider two problems: reconstructing an original binary from encoded bit strings and transforming an infix expression into a postfix one. This paper proposes optimal parallel algorithms for these problems based on parentheses matching using linear arrays with optical buses. The proposed algorithms run in a constant number of communication cycles, using processors equal to the input size. The main contribution of this paper is to design cost optimal algorithms with a constant time for these problems.

Key words:
Optical buses, Parentheses matching, Binary tree, Infix/postfix expression

1. Introduction

A large body of research has recently been devoted to the architectures and algorithm development for optically interconnected parallel computer systems [1-9]. Optical interconnected parallel architecture can offer advantages over electronic counterparts including high connection density and relaxed bandwidth-distance product. Parallel systems with optical interconnections also resolve some limitations of electronic buses such as limited bandwidth, capacitive loading, and cross-talk.

Unlike electronic buses on which signal propagation is bidirectional, optical channels are inherently unidirectional and have predictable delay per unit length. Some problems such as sorting [1, 3, 4], routing [4], image transformation [5], graph [6], permutation [7], and matrix multiplication [9] have been solved efficiently on parallel architectures using optical buses. In this paper, we use the linear array of processors with slotted optical buses (for short, LASOB) as a computation model.

The parentheses matching problem is to determine the index of the mate for each parenthesis, and plays an important role in the design of parallel algorithms such as graph and text matching. Shen[10] showed that some graph and tree problems can be reduced to the parentheses string representation. This paper includes two problems as follows: reconstructing an original binary from encoded bit strings and transforming an infix expression into a postfix one. We present optimal parallel algorithms for these problems based on parentheses matching using linear arrays with optical buses.

In computer applications, binary tree is a data structure consisting of an array of records, where each record corresponds to a tree node and contains the data it carries, and pointers to the parent and its two children. In some applications the binary tree is encoded as a sequence of several integers, and a variety of encoding techniques are proposed in [11, 12]. One of the most commonly used encoding techniques is the bit-string code proposed in [11].

The reverse process, reconstructing the binary tree data structure from its sequence of integers, is referred to as the decoding. Given a *2n* bit-string code (where *n* is the number of nodes in the binary tree), the original binary tree of *n* nodes must be decoded from such an encoded bit-string code.

In [12, 13] parallel algorithms for two problems are given, both running in *O(logn)* time using *O(n/logn)* processors in the EREW PRAM model. These algorithms are optimal in the sense that the product of time and number of processors is asymptotically the same as the optimal sequential time. However, both algorithms are not performed in a constant time. A constant time algorithm for decoding a binary tree from encoded bit strings was described on the BSR computation model, which consists of *n* processors sharing *m* memory locations and having broadcast instructions [14]. Compared with the BSR model, our parallel computation model has no shared memories and each processor has just a constant number of memory locations.

This paper proposes optimal parallel algorithms for both problems based on the parentheses matching, which run in a constant number of communication cycles on linear arrays with slotted optical buses using processors equal to the input size. The main contribution of this paper is to

Manuscript received April 5, 2007
Manuscript revised April 25, 2007

design cost optimal algorithms with a constant time for these problems.

The reminder of this paper is organized as follows. In the next section the architecture of the LASOB is formally described, and several basic operations which will be used in our algorithms are introduced. Constant time algorithms for decoding the binary tree and transforming the postfix expression are described in the section 3 and 4. Finally, we conclude the paper in section 5.

2. The LASOB model

2.1 Architecture

Parallel systems with optical interconnections can be considered as an alternative to conventional parallel systems because optical buses have high connection density and relaxed bandwidth-distance product over electronic buses. These architectures with optical buses include RASOB [1], APPB [2], AROB[4], etc.

The basic architecture of the LASOB to be introduced in this paper is the same as the 1-dimensional RASOB model, and is very similar to the APPB model. Since each processor on the LASOB is not allowed to reconfigure its internal port connections, the architecture of the LASOB has much lower switch control than other reconfigurable meshes.

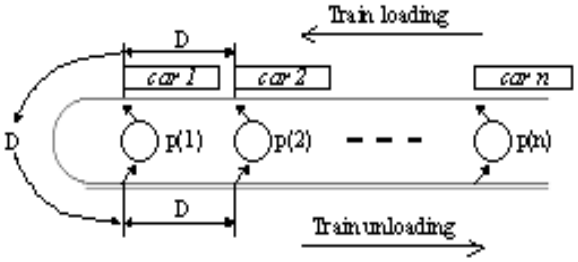

Figure 1. The LASOB model

The architecture of the LASOB is shown in Figure 1. As shown in Figure 1, the bus is divided into two segments interconnecting the linear array of processors: upper segment(train loading) and lower segment(train unloading). Each processor has a transmitting interface to the upper segment and a receiving interface to the lower segment. Thus, the processors write packets to the upper segment, and read packets from the lower segment. During a bus cycle, multiple processors can transmit their packets by using different time slots of the bus. This is possible because optical buses are inherently unidirectional, and have predictable delay per unit length.

In Figure 1, the numbering of the processors is denoted from left to right by *P(1), P(2), ... , and P(n)*, respectively. Each processor on the upper and lower bus segments is separated in time by $D=bw+d$ (seconds) from its neighbors, where b is the maximal length of a packet in bits, and w is the optical pulse width (or bit duration) in seconds. Also, $d>0$ is used as guard bands to tolerate synchronization error to a certain degree. This temporal separation can be achieved by separating the two neighboring transmitter (receiver) interfaces on the upper (lower) segment with a fiber length $D \times c$, where c is the speed of light in the fiber.

End-to-end propagation is defined as a communication cycle. This paper assumes that at the beginning of a communication cycle, a train of n cars (slots) is originated at the right most end of the upper segment bus. Each car can be regarded as an empty packet slot with the duration of D and is numbered 1 through n from left to right. Then, a simple assignment of the cars is to let processor *P(1)* use car 1 for sending its packet, let *P(2)* use car 2 for sending its packet and so on.

With this assignment of the cars, the time when the processor *P(i)* may transmit its packet, relative to the beginning of the communication cycle, is given by $(n-1)D$. More specifically, if the processor *P(i)* is expecting a packet sent by *P(j)*, it can calculate the time it should pick up the packet by $(n+i+j-2)D$. All transmissions are synchronized and each processor can send and receive packets at specific time stated above. A communication cycle also is assumed as a constant time, since it is compared to one computation in a reasonable size (almost 1,000 processors) [2].

2.2 Basic operations

We first define several basic operations that will be used throughout the paper. *P(i)*, $1 \le i \le n$, represents the *i-th* processor on an LASOB with n processors.

Lemma 1. *In an LASOB of size n any permutation can be routed in a constant number of cycles.*

Proof. Let $\Pi(1), \Pi(2), \cdots, \Pi(n)$ be a permutation of 1, 2, ..., n. Suppose that for all i, $1 \le i \le n$, each processor *P(i)* wants to send its packet to the processor $P(\Pi(i))$. If the address of the destination processor is known to the source processors, this type of one-to-one communication can be done in a communication cycle in parallel by assigning a packet of the *i-th* processor to car $\Pi(i)$. This technique is called time-division destination-oriented multiplexing.

Given a binary sequence $b_1, b_2, \cdots, b_n$, the *prefix sums* of a binary sequence compute $b_1 + b_2 + \cdots + b_j$ for each j, $1 \le j \le n$.

Lemma 2. *Consider an LASOB which consists of n processors. If each processor has a bit value, then the prefix sums of these values can be computed in a constant number of cycles [2, 5].*

The *parentheses matching problem* is to determine the pair of each parenthesis within a balanced string of *n* parentheses. A string *s* consisting of left and right parentheses is to be balanced if it contains the same number of left parentheses and right parentheses and it satisfies the prefix property, namely, that in every prefix of *s* the number of right parentheses does not exceed the number of left parentheses. Given a balanced string *s* consisting of left and right parentheses, the parentheses matching problem asks to find all pairs of matched parentheses in *s*.

Lemma 3. *If each processor has a left or a right parenthesis, the parentheses matching problem can be solved in a constant number of cycles.*

Proof. We proved Lemma 3 in details in [15].

3. Reconstructing a binary tree from encoded bit strings

Binary tree is a data structure that at most has two children nodes and one parent node, and is used in many computer applications. In some applications such as encryption fields, binary tree can be encoded as a sequence of bits by a variety of encoding techniques. The reverse process, reconstructing the original binary tree from its encoded sequence, is referred to as the decoding. In this section, a parallel algorithm for decoding the original binary tree from its encoded bit strings is introduced, using an LASOB with processors equal to the input size.

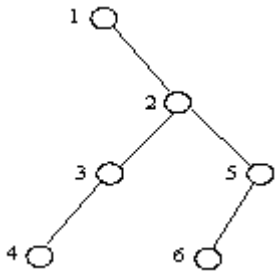

Figure 2. A binary tree

We first consider the binary tree as shown in Figure 2. Let us define the binary tree of Figure 2 as *T*. All nodes of the binary tree *T* in Figure 2 are assigned 1 and all missing children are replaced with real ones but these new nodes are assigned 0. The binary tree assigned in such a way is defined as the extended binary tree *T'* (see Figure 3).

The encoding sequence is the same as the preorder traversal of such an extended binary tree, but the last 0 is omitted. Thus, an encoded bit-string code for *n* nodes consists of *2n* bits. For an example, the binary tree of Figure 3 is encoded 101110001100. An *O(logn)* time algorithm for decoding a binary tree is proposed in [13]. In this paper, a constant time algorithm is described on an LASOB.

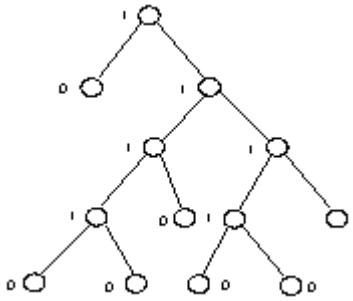

Figure 3. Extended binary tree

The parentheses matching problem is used as a basic operation in designing our algorithm on an LASOB. We can transform an encoded bit-string into a string which consists of left and right parentheses. When a bit-string, 101110001100, is given as input, if 1 is replaced by a left parenthesis and 0 by a right parenthesis, the encoded bit-string can be represented as '()((()))(())'.

Let *l(i)* be the position of the *i-th* 1 (left parenthesis) in a bit-string of length *2n*, and let *r(i)* be the position of its matching 0 (right parenthesis). For an example, in 101110001100, we have *l(1)=1, l(2)=3, l(3)=4, l(4)=5, l(5)=9, l(6)=10, r(1)=2, r(2)=8, r(3)=7, r(4)=6, r(5)=12, r(6)=11*. These values are to be used in determining the left and right child of a node in the binary tree. Also, this procedure satisfies the following two properties.

Property 1. *If r(i) is equal to l(i)+1, the left child of a node i does not exist, and otherwise the left child of a node i is the (i+1)-th 1.*

Property 2. *If the value of the (r(i)+1)-th bit is 0, the right child of a node i does not exist, and otherwise the right child of a node i has a node j such that l(j)=(r(i)+1).*

The properties 1 and 2 can be verified easily because an encoded bit-string is constructed using the preorder traversal and this bit-string is reduced to the parentheses matching problem. In case of the property 1, the left child of a node *i* is appeared in the immediate next position of the *l(i)-th* bit in an encoded bit-string according to the

inherent property of the preorder traversal, and if the value of this position is 0, the left child of a node *i* does not exist and otherwise the left child for a node *i* is set to *i+1*.

We now look into the meaning of the property 2, which is used to determine the right child of a node in the binary tree. By the preorder traversal, the right child of a node *i* is visited after traversing first all nodes consisting of its left subtree. In an encoded bit-string, the corresponding bit position of a node *i* is *l(i)*, and bits corresponding to all nodes of the left subtree of a node *i* are ended at the bit position *r(i)*.

Thus, the right child of a node *i* can be determined by checking the position of the *(r(i)+1)-th* bit in an encoded bit-string, and if the value of this position is 0 the right child of a node *i* does not exist and otherwise the right child of a node *i* is a node *j* such that *l(j)=r(i)+1*.

We next will describe how to implement the original binary tree from an encoded bit-string on an LASOB, using two properties presented above. Let *lchild(i)* be the left child of a node *i*, and let *rchild(i)* be the right child of a node *i*. Initially, it is assumed that an encoded bit-string of length *2n* is given as input, and the value of the *i-th* bit position in the input string is stored in the processor *P(i)* on an LASOB with *2n* processors. A brief description of our algorithm is given below, and it consists of 2 phases.

Phase 1. Compute *l(j)* and *r(j)*, $1 \le j \le n$, from the input string.

Phase 2. Determine the left and right child of a node *j* in the binary tree by the property 1 and property 2, respectively.

In phase 1, each processor *P(i)*, $1 \le i \le n$, computes *l(i)* and *r(i)* from an encoded bit-string. In phase 2, the left and right child of each node in the binary tree is determined. The detailed proofs of each phase are described in Theorem 1.

Theorem 1. *Given an encoded bit-string of length 2n, the problem of decoding the binary tree from such a bit-string can be solved in a constant number of cycles on an LASOB with 2n processors.*

Proof. Each phase in the algorithm is proved as follows:

Phase 1. The parentheses matching problem described in Lemma 3 is used as a basic operation. In order to construct a string which consists of left or right parentheses from an encoded bit-string, each processor *P(i)*, $1 \le i \le 2n$, assigns itself a left parenthesis if its bit value is 1 and otherwise a right one.

Let *m(i)* be the index of the processor that has the matching pair of a left or a right parenthesis stored in the processor *P(i)*. Then *m(i)*, $1 \le i \le 2n$, can be easily computed in a constant number of cycles by applying the parentheses matching algorithm in Lemma 3.

Let *psum(i)* be the prefix sum of the *i-th* bit position in an encoded bit-string of length *2n*. Then each processor *P(i)* can compute *psum(i)* in a constant number of cycles by Lemma 2. We now show how to efficiently determine *l(i)* and *r(i)* in each processor *P(i)*. Each processor *P(i)* sets *l(psum(i))* to *m(i)* if it has a left parenthesis or *r(psum(i))* to *m(i)* if it has a right one. Because each processor *P(i)* already has *psum(i)*, the processor *P(i)* can compute *l(i)* and *r(i)* in a constant number of cycles.

Finally, by Lemma 1, each processor *P(i)* routes *l(psum(i))* to the processor *P(psum(i))* if it has a left parenthesis or *r(psum(i))* to *P(psum(i))* if it has a right one. Then, the processors *P(i)s*, $1 \le i \le n$, from leftmost to right have *l(i)* and *r(i)*.

Phase 2. This phase computes *lchild(j)* and *rchild(j)* for each node *j*, $1 \le j \le n$, using *l(j)* and *r(j)* computed in phase 1. After completing phase 1, each processor *P(j)*, $1 \le j \le n$, has both *l(j)* and *r(j)*. The left and right child of each node can be determined using these values and an encoded bit-string given initially as input. First, we show how to determine the left child of each node by the property 1. Because each processor *P(j)*, $1 \le j \le n$, has *l(j)* and *r(j)* by the phase 1, the processor *P(j)* can set *lchild(j)* to *NULL* if *r(j)=l(j)+1* and otherwise *lchild(j)* to *j+1*. It is possible to perform in a constant number of cycles as this procedure can be computed directly in each processor.

We next show how to determine the right child of each node by the property 2. Each processor *P(j)*, $1 \le j \le n$, sets *rchild(j)* to *NULL* if the value of the *(r(j)+1)-th* bit position is 0. This is possible because the value of the bit stored in the processor *P(r(j)+1)* can be routed to the processor *P(j)* in a constant number of cycles by Lemma 1 and all the *r(j)s* are distinct.

Now, when the value of the *(r(j)+1)-th* bit position is 1, we show how each processor *P(j)* sets *rchild(j)* to *k* such that *l(k)=r(j)+1*. Each processor *P(j)*, $1 \le j \le n$, can check whether the value of the *(r(j)+1)-th* bit position is 1 or 0. To find the value of *k* such that *l(k)=r(j)+1*, each processor *P(k)*, $1 \le k \le n$, routes *(l(k), k)* to the processor *P(l(k))* by Lemma 1. Then the processor *P(l(k))* has the index *k* sent, and the processor *P(j)*, $1 \le j \le n$, can obtain the index *k* satisfying this condition (i.e., *l(k)=r(j)+1*) from the processor *P(r(j)+1)*, where $1 \le l(j), r(j) \le 2n$. Thus the processor *P(j)* can set *rchild(j)* to the index *k* received from the processor *P(r(j)+1)* when all conditions are

satisfied. These steps can be solved in a constant number of cycles by applying Lemma 1 because only simple routings among the processors are needed.

4. Transforming an infix expression to a postfix one

In this section, we present an optimal parallel algorithm for transforming an infix expression into a postfix one on an LASOB. In computer science, the postfix expression is widely used in calculating arithmetic expressions using stack operations. This paper considers only the following operators: +, -, *, /, ↑ . However, the algorithm proposed in this paper can be easily extended to arithmetic expressions with general operators. In what follows our algorithm consists of four phases and we later will prove how each phase is implemented in a constant number of cycles on an LASOB.

Phase 1. Insert the parentheses to an infix expression.
Phase 2. Determine the matching pairs of all parentheses using the parentheses matching algorithm.
Phase 3. Move each operator of an infix expression to the position of the nearest right parenthesis enclosing it.
Phase 4. Remove all parentheses and then rearrange the operands and operators only.

Theorem 2. *Given an infix expression of length n, the problem of transforming an infix expression into a postfix one can be solved in a constant number of cycles on an LASOB with n processors.*

Proof. The detailed proofs of each phase in the algorithm are given below.

Phase 1. Assume that an infix expression of length *n*, x_1, x_2, ⋯, x_n, is given and each x_i, $1 \le i \le n$, is stored in the processor *P(i)* on an LASOB with *5n* processors. In other word, the given infix expression is stored in consecutive positions on *n* leftmost processors. x_i has one of the following symbols: operator, operand, left parenthesis, or right parenthesis.

Phase 1 can be implemented efficiently with the help of the parentheses insertion algorithm of Knuth [16]. Each processor *P(i)*, $1 \le i \le n$, with operator inserts parentheses to itself according to the rules as follows: '))x_i((' if x_i is '+' or '-', ')x_i(bb' if x_i is '*' or '/', and '(($x_i$bb' or ' x_i))bb' if x_i is '(' or ')', respectively, where *b* is a blank. Also, each processor *P(i)* with operand assigns ' $x_i$bbbb' to its processor, and then *n* leftmost processors have 5 symbols each.

The symbols on *n* leftmost processors are rearranged on *5n* processors so that each processor has only a symbol. This rearrangement can be solved easily in a constant number of cycles by Lemma 1. Next, each processor on *5n* processors assigns 0 if it has symbol 'b' and otherwise 1, and then we compute the prefix sums of the binary strings by Lemma 2. By Lemma 1, each processor without symbol 'b' route its symbol to the processor corresponding to the value of its prefix sum. Hence, it is obvious that phase 1 is performed in a constant number of cycles.

For an example, we consider an infix expression as follows: *a / b - c + d ↑ (e / f + g ↑ h)*. After completing phase 1, the input expression is transformed into the below expression, which includes balanced parentheses enclosing all operators.

(((a / b) - c) + (d ↑ ((e / f) + (g ↑ h))))

Phase 2. In phase 2, the matching pairs of all parentheses from the result of phase 1 are determined. The parentheses matching problem can be solved in a constant number of cycles by Lemma 3. Let *m(i)* be the index of the matching pairs for left or right parentheses. Then the value of *m(i)* in the processor *P(i)* with a left or a right parenthesis can be obtained from Lemma 3 and this value later will be used in phase 3.

Phase 3. Each operator in the result expression obtained from phase 2 is moved to the position of the nearest right parenthesis enclosing it. This routing is performed as follows: First, all the processors *P(i)s* with the result of phase 1 route their operators to the processors *P(i+2)s* if the processors *P(i+2)s* have a right parenthesis. Next, all the processors *P(i)s* route their operators to the processors *P(m(i+1))* if the processors *P(i+1)s* have a left parenthesis. These steps can be completed in a constant number of cycles by Lemma 1 because only simple routings are needed.

Phase 4. In this phase, all parentheses are removed and then the remaining operands and operators are to rearrange in consecutive processors from leftmost to right on an LASOB. These steps also are done in a constant number of cycles as follows: First, the prefix sums in the binary sequence are computed by Lemma 2 after assigning 0 if a parenthesis is removed and otherwise 1, and then both operands and operators are routed in the processors corresponding to the index equal to the values of their prefix sums. After completing this phase, the input expression in the above example can be converted as the following postfix one.

a b / c - d e f / g h ↑ + ↑ +

Finally, as phases 1, 2, 3, and 4 can be computed in a constant number of cycles, and our algorithm performed on *5n* processors can be simulated easily in a constant number of cycles on *n* processors, the theorem is proved.

5. Conclusions

In this paper, we considered two problems as follows: reconstructing an original binary from encoded bit strings and transforming an infix expression into a postfix one. This paper proposed efficient parallel algorithms for these problems based on parentheses matching using linear arrays with optical buses. The proposed algorithms run in a constant number of communication cycles using processors equal to the input size. In the sense of the product of time and the number of processors used, all of our algorithms are both time and cost optimal.

References

[1] Hamdi, C. Qiao, Y. Pan, and J. Tong, "Communication-efficient sorting algorithms on reconfigurable array of processors with slotted optical buses," J. of Parallel Distribut. Comput. 57, pp. 166-187, 1999.

[2] S. Pavel and S. G. Akl, "On the power of arrays with reconfigurable optical buses," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications*, Vol. III, pp. 1443-1454, 1996.

[3] S. Pavel and S. G. Akl, "Integer sorting and routing in arrays with reconfigurable optical buses," *Proc. Int'l Conf. on Parallel Processing*, Vol. II, pp. 90-94, 1996.

[4] C. H. Wu, S. H. Horng, Y. R. Wang, and H. R. Tsai, "Optimal geometric algorithms for digitized images on arrays with reconfigurable optical buses," *Microprocessors and microsystems*, Vol. 30, pp. 425-434, 2006.

[5] M. Kim, "Efficient transformations between binary images and quadtrees on a linear array with reconfigurable optical buses," *The Trans. of KIPS*, Vol. 6, pp. 1511-1519, 1999.

[6] K. Y. Pan and M. Hamdi, "Solving graph theory problems using reconfigurable pipelined optical buses," *Parallel Computing*, Vol. 26, pp. 723-735, 2000.

[7] J. L. Trahan, A. G. Bourgeois, Y. Pan, and R. Vaidyanathan, "Optimally scaling permutation routing on reconfigurable linear arrays with optical buses," *J. of Parallel Distribut. Comput.* 60, pp. 1125-1136, 2000.

[8] C. Wu, S. J. Horng, and H. R. Tsai, "Efficient parallel algorithms for hierarchical clustering on arrays with reconfigurable optical Buses," *J. of Parallel Distribut. Comput.* 60, pp. 1137-1153, 2000.

[9] K. Li, Y. Pan, and S. Q. Zheng, "Parallel matrix computations using a reconfigurable pipelined optical bus, "*J. of Parallel Distribut. Comput.* 59, pp. 13-30, 1999.

[10] W. Shen, "Data structures for parallel processing," *Http://carbon.cudenver.edu*, 2007.

[11] S. Zaks, "Lexicographic generation of ordered trees," *Theoretical Computer Science*, vol. 10, pp. 63-82, 1980.

[12] S. Olariu, J. L. Schwing, and J. Zhang, "Optimal parallel encoding and decoding for trees," *Int'l J. Foundation of Computer Science*, Vol. 3, pp. 1-10, 1992.

[13] E. Dekel and S. Sahni, "Parallel generation of postfix and tree forms," *ACM Trans. Programming Languages and Systems* 5, pp. 300-317, 1983.

[14] Stojmenovic, "Constant time BSR solutions to parenthesis matching, tree decoding, and tree reconstruction from its traversals," *IEEE Trans. Parallel and Dist. Systems*, Vol. 7, pp. 218-224, 1996.

[15] Y. H. Kim, "An optimal parallel algorithm for generating computation tree form on linear array with slotted optical buses," *J. of KISS: computer systems and theory*, Vol. 27, pp. 475-484, 2000.

[16] D. E. Knuth, *The art of computer programming: Fundamental algorithms*, Addison-Wesley, Reading, MA, 1973.

Young-Hak Kim received the M.S. and Ph.D. degrees in Computer Engineering from Sogang University in 1989 and 1997, respectively. He is currently an associate professor in the school of computer and software engineering at Kumoh National Institute of Technology, Gumi, Korea. He was a visiting scholar in the school of electrical and computer engineering at Georgia Institute of Technology. His research interests include parallel algorithm, parallel processing, and embedded system.