論 文

A Parallel Emulation Scheme for Data-Flow Architecture on Loosely Coupled Multiprocessor Systems

Yong Doo Lee*, Soo Hwan Chae** *Regular Members*

弛緩 結合形 多重 프로세서 시스템을 使用한 데이터 플로우 컴퓨터 구조의 竝列 에뮬레이션에 關한 硏究

正會員 李 龍 斗* 正會員 蔡 洙 煥**

Abstract

Parallel architecture based on the von Neumann computation model has a limitation as a massively parallel architecture due to its inherent drawback of architectural features. The data-flow model of computation has a high programmability in software perspective and high scalability in hardware perspective. However, the practical programming and experimentation of data-flow architectures are hardly available due to the absence of practical data-flow architectures, while a number of von Neumann parallel machines are available. In this paper, we present a programming environment for performing the data-flow computation on conventional parallel machines in general, loosely coupled multiprocessor systems in particular. We build an emulator for tagged token data-flow architecture on the iPSC/2 hypercube, a loosely coupled multiprocessor system. The emulator is a shallow layer of software executing on an iPSC/2 system, and thus makes the iPSC/2 system work as a data-flow architecture from the programmer's viewpoint. We implement various numerical and non-numerical algorithms in a data-flow assembler language, and then compare the performance of the program with those of the versions of conventional C language. Consequently, we verify the effectiveness of this programming environment based on the emulator in experimenting the data-flow computation on a conventional parallel machine.

요 약

노이만 계산 모델의 병렬처리 구조는 구조 속성상의 취약성으로 인해 대량 병렬처리 구조로서는 한계가 있다. 데이터 플로우 계산 모델은 소프트웨어적 고 프로그램성과 하드웨어적 높은 개발 가능성을 갖고 있다. 그러나 실제 데이터 플로우 구조에서는 프로그래밍과 실험을 행하고자 할때, 노이만 방식의 기계는 많지만 실제 데이터 플로우 컴퓨터가 없으므로 대단히 어렵다. 본 논문에서는 일반적 재래 병렬처리기계중 하나인 이완결합 다중프로세서 시스템위에서 데이터 플로우 방식의 계산을 수행시킬 수 있는 프로그래밍

* 大邱大學校 電子工學科
Dept. of Electronic Eng., Taegu Univ.

** 韓國航空大學校 電子計算學科
Dept. of Computer Eng., Hankuk Aviation Univ.

論文番號 : 93-190

환경을 제시하였다. 에뮬레이터는 iPSC/2 하이퍼 큐브를 이용하여 Tagged Token 데이터 플로우 구조를 구축하였다. 본 에뮬레이터는 iPSC/2 시스템에서 소프트웨어적 박층 실험이므로 프로그래머의 입장에서는 iPSC/2 시스템이 데이터 플로우 구조로서 동작하는 것으로 간주한다. 여러가지 수치 혹은 비수치 알고리즘을 데이터 플로우 어셈블리어로 구현하여 재래식 C 언어에 의한 것과 프로그램의 성능을 비교하였다. 이로써, 재래식 병렬처리 기계상의 에뮬레이터를 통한 실험적 데이터 플로우 계산을 행할때 이 프로그래밍 환경의 효율성에 대하여도 검정하였다.

Ⅰ. Introduction

Device technology is expected not to guarantee the linear increase in its switching speed, whereas a state of the art computer do not meet the high computing power required in scientific computation. Parallel processing is considered a promising approach for coping with the demmands of computation. In order to enable parallel processing, a number of parallel machines based on the von Neumann computation model have been built. However, the inherent sequential nature of the model prevents massively parallel architectures. Those machines in general suffer high communication latency and poor programmability since users must face low level architectural features in programming applications[1].

The data-flow model of execution is provided as a solution to the von Neumann bottleneck, thus envisioning true parallel processing. Dynamic instruction scheduling, based on the data-driven principles of the model, provides a potential to hide communication latency, resulting in the exploitation of fine-grain parallelism[2]. The functional semantics of the data-flow model allows the data-flow mode to be a viable alternative to address the issue of programmability of multiprocessor systems. Consequently, given the limitations of this explicit parallelism approach, data-flow is more robust and versatile in that it will provide complete transparency to the user and high accessibility to large-scale parallel machines. Indeed, the data-driven approach is intrinsically scalable.

This work demonstrates that data-flow architecture has superior programmability from the programmer's point of view in conventional sequential multiprocessor systems and higher processor utilization. The execution time of numeric and non-numeric algorithm both on the data-flow machine and the conventional sequential multiprocessor system was compared. We selected the INTEL iPSC/2 hypercube system as a prototype distribution message passing multiprocessor system. We implemented a macro actor tagged-token data-flow machine emulator on the iPSC/2 hypercube system.

Ⅱ. Background and Emulation Model

2.1 The programmability issue

In a conventional von Neumann programming enviroment, it is essentially the programmer's responsibillity to ensure that the memory latencies are masked by the execution of other tasks and that the tasks are appropriately synchronized amongst themselves. This is the issue of programmability of multiprocessor systems. In fact, maintaining a large number of tasks active among multiple processors in an intricate configuration, and synchhronizing them in proper order to obtain a safe execution, is a humanly impossible feat. Indeed, the problems of task partitioning for large multiprocessor systems (greater than two) are extremely difficult to surmount. In the conventional models of execution, the programmer must be aware of the potential parallelism in the program as well as of all the synchronization requirements so that the program can be partitioned, allocated, and sequenced safely, i.e.,

produces the same results as if the program had been run on a single processor machine. Since there are many potential points of synchronization to be mapped over a large number of pairs of processors, safe programming can only be guaranteed by a very conservative approach to dependencis would be introduced. In order to fully utilize the potential of multiprocessor technology, a different approach to programming is needed altogether. Such as the functional model of execution in which instructions are functions to which arguments are applied. This solution has the advantage of embedding in program the data dependencies, thereby relieving the programmer from this burden.

2.2 Data-driven : A Functional Model of Execution

The degree of explicit control of the scheduling of operations on hardware resources characterizes the very principles of computation models. The pure data-driven computation model and the von Neumann model are the two extreme of the continuum in the scheduling perspective. The scheduling of operations in the von Neumann computation model is static, i.e. the execution sequence is decided at compile time, while the scheduling in a pure data-driven computational model is dynamic upon availability of operand data.

In the data-driven computational model, a program is represented as a data-flow graph(Fig. 1), a digraph with an explicit data-dependency relation by arcs as well as computation by nodes and implicit semantics of data-driven execution. The major impact of the data-driven model is the capability to hide the latency, and to eliminate explicitly synchronization by dynamic scheduling based upon data availability, while maximally exposing available parallelism[2],[3]. In short, the data-flow model of execution has been designed in order to solve two of the most important problems of multiprocessor systems design :

·High memory latency : a large scale multiprocessor should contain many processors which can be widely separated. This implies that communication costs for synchronization between processors are very heavy and will add a considerable penalty to data transfers.

·High context switching time and low processor utilization : all processors in multiprocessor constitute a long pipeline(including the internal pipeline segments of the processor themselves, if applicable). In order to keep this pipeline operating at maximum efficiency, a large unmber of jobs must be kept active simultaneously. This problem is further complicated by the fact that task switching between processors may be very high.

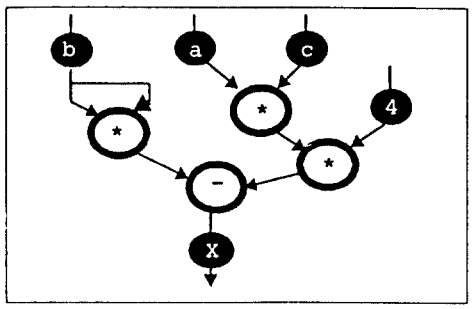

Fig 1. An example data-flow graph for $b^2 - 4ac$

2.3 The Macro-actor data-driven model

Even though the data-flow computation model has a potential to hide the latency and to expose available parallelism, the model in its pure form would not be acceptable due to expensive hardware cost and the scheduling overhead. In fact, dynamic scheduling is not necessary for the instruction in a sequential thread, only introducing scheduling overhead, i.e., cost of machine operands and waste of system resources in passing operands without registers. Overall, compared to the von Neumann machines, the data-flow architectures costs are expensive due to the hardware supports for the execution control, e.g. operand matching facility.

The von Neumann model, on the other hand, is

fragile at the unpredictable operations due to its static scheduling. Therefore, the stackness of the control at a point between the two extremes of scheduling is desirable to resolve the bottleneck. As an alternate, a macro actor data-driven model was proposed[4][5]. The model reduces the overhead from dynamic scheduling at the micro level by clustering a set of sequential instruction into an actor executed in sequence, while it preserves the benefits of dynamic scheduling by maintaining the principle of data-driven execution at the macro level.

2.4 The dataflow model for emulation

The target dataflow model for emulation is a tagged token macro actor dynamic data-flow architecture(MATTDA) that has been heavily simulated in USC(University of Southern California) data-flow researach group. The MATTDA is a data-flow/von Neumann hybrid model proposed to reduce the synchronization overhead and exploit the locality among instructions[3],[6]. The scheduling unit is a collection of instruction clustered according to performance criteria at compile time.

As shown in Fig.2, the MATTDA consists of identical processing elements(PE) connected by a packet network with a global I-structures for structured data. The basic structure of each PE is the same as the MIT tagged token data-flow architecture ; a four-stage pipelined architecture which consists of Match unit, Fetch unit, ALU unit and Output unit.

In the MATTDA model, a token can carry "fat" data such as vectors. The Vector data of incoming tokens are stored in the Data store and only the data frame pointer or vector data are sent to ALU which can directly address the matched vector data.

A von Neumann execution model is currently assumed for the ALU. As in conventional assembly language, the micro instruction within the ALU has three addressing modes ; immediate, direct and indirect addressing. Each datum of vector token data can be referenced by its port unmber and its displacement from the vector

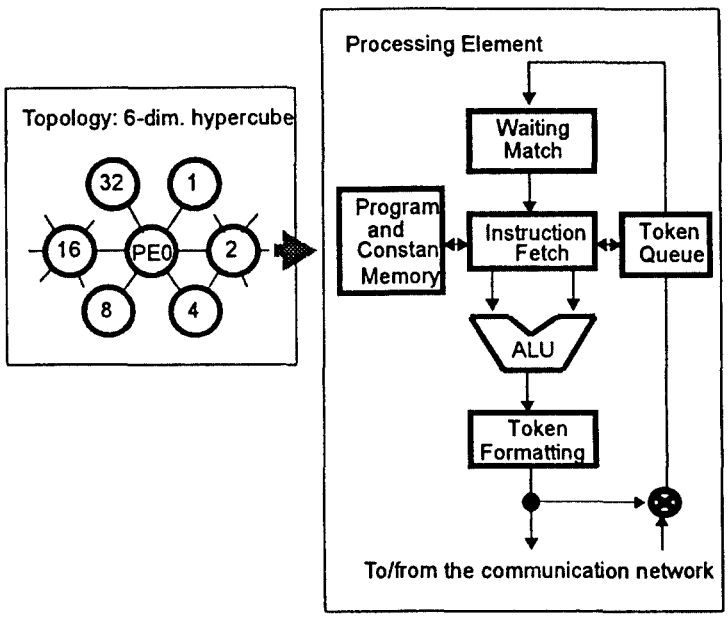

Fig 2. The data-flow model for emulation

head. After processing the macro data, the ALU should indicate the range of valid data for producing new tokens with a tag to the OUTPUT facility.

Operationally, on each PE, every incoming token is enqueued in Match queue implemented by associative memory or hash queues in general. According to the associative or hashing function, tokens are matched with the tags. Once all the input tokens for an actor are matched, an activity for the actor is formed and then enqueued into the Enable actor queue. Later, when the actor is scheduled, the instructions inside the actor are sequentially executed, finally generating a number of output tokens. These tokens are again routed to the destination PEs. Consequently, we see that two levels of scheduling is employed in the MATTDA : data driven scheduling at the inter-actor level and control-driven scheduling of von Neumann model at intra-actor operations.

Ⅲ. The mapping the MATTDA to iPSC/2 system

3.1 iPSC/2 Architecture

Intel's iPSC/2 Concurrent Supercomputer is the cost effective solution for large-scale applications. Typical application include computation : mechanics, petroleum exploration, electronic design, molecular modeling, and tactical and strategic systems. In and iPSC/2 system, a large number of processors or nodes work concurrently on the parts of a single problem[7].

An iPSC/2 system consists of compute nodes, I/O nodes, and a front-end processor called the host. A nodes is a processor/memory pair. Its physical memory is distinct from that of the host and other nodes. Each node runs the NX/2 operating system, uses message passing to communicate with other nodes, and can access both the host file system and the iPSC/2 Current File System.

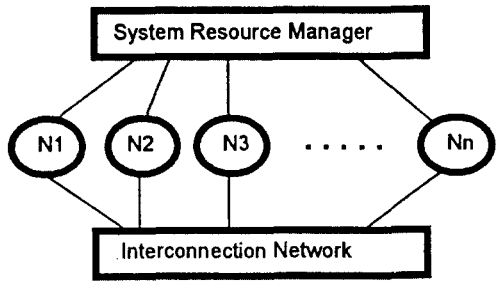

Fig 3. iPSC/2 system

3.2 The communication in iPSC/2 system

iPSC/2 support both synchronous and a synchronous communication among PEs. In the current programming environment, a number of C library function are provided to support those two communications. The synchronous communication enforces the process which calls a communication function, i.e., to send or receive a message, to be suspended until the requested function finishes. The functions for sending and receiving a message in C language are send( ) and recv( ), repectively.

The asynchronous communication enable the computation and communication to be overlapped. The process which calls an asynchronous send or receive function does not have to wait until the communication for the message finishes. This is more efficient in execution time than the synchronous communication. The functions for asynchronously sending and receiveing a message in C language are isend( ) and irecv( ), repectively. Library functions to secure mutual exclusion for critical sections which is indispensable in asynchronous communication are provided.

In addition to general communication facilities as noted previously, there is an asynchronous communication function in which the user can install a service routine in for a messages of a type. When a message arrives arrives at a a PE, the system is trapped to the service routine for the type of the message which was the install by the user. This capability is very useful where a reg

ular manipulation for all messages of a type is require.

3.3 Mapping the emulated model to the iPSC/2

The main objective of the emulator is to make an iPSC/2 system work as the MATTDA by providing a shallow layer of software on iPSC/2 Though the hardware of an iPSC2 does not provide the exact facilities for the MATTDA, we can utilize some of them in our emulator. So, we attempt to map the components of the target emulation system to the iPSC/2 system in order to maximally use physical components. With this objective in mind, we map each node in the iPSC/2 into a processing element(PE) of the MATTDA system by locating processes corresponding to major function units. The interprocessor communication facilities of the iPSC/2 are just mapped into the interconnection network among PEs. The system resource manager which consists one 32 bit microprocessor, main memory and UNIX as a operating system is mapped to the system manager processor of the MATTDA[4],[7].

Ⅳ. The Overall structure of the Emulator

The emulator consists of two kinds of processes. The HOST process running on the System Resource Manager of iPSC/2 system emulates the general function of System Resource Manager of the MATTDA. As the PEs in the MATTDA are functionally homogenous, one single process named the NODE process is provided for each node in the iPSC/2 system. The NODE process emulates the function of hardware components of a PE and I-structure controller in the MATTDA.

4.1 The HOST process

The key role of the system resource manager of the MATTDA is to provide the user interfaces. On the system management side, it also provides facilities for downloading of dataflow program graphs and configuration information, and input/output interfaces during execution. The HOST process has the corresponding software components to emulate the functions of the system resource manager. Whenever a program is executed, the HOST process loads the NODE process on each

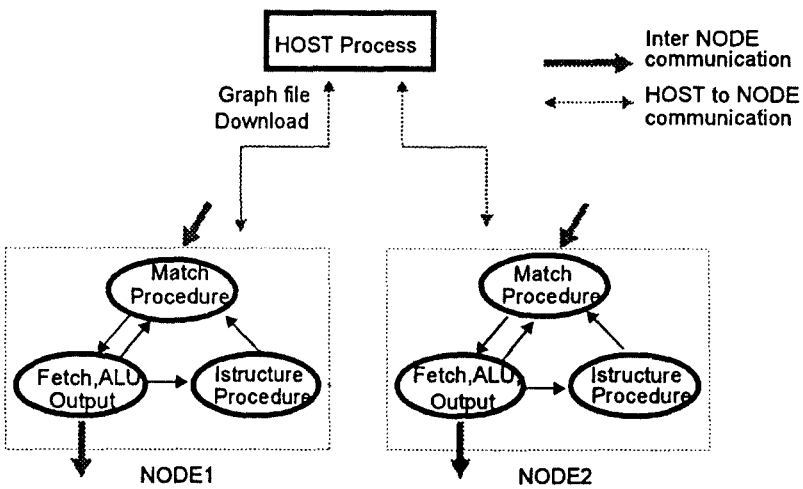

Fig 4. Mapping of IPSC/2 system to emulation model

node allocated to the user. Then, the HOST process sends the program graph to a NODE processes of the nodes through a message with appropriate inormation for system configuration. During the process of the program execution, the HOST interacts with nodes to service the I/O functions. Finally, when the program has terminated, the HOST process collects the execution result and then releases the NODE processes from the nodes.

4.2 The NODE process

A NODE process resides on each node of the iPSC/2 system, emulating a variety of function of a PE in the MATTDA system. The NODE processes consist of a number of engines for the mathch, fetch, alu and output facilites of the MATHCH, FETCH, ALU and output facilites of the MATTDA. For input tokens, the engines operates with an appropriate function. Finally, tokens generated by a PE are routed to appropriate PEs according to mapping rules. The routing of tokens in NODE process is done by using the asynchronous communication facilities in the iPSC/2 system.

4.2.1 PARSE engine

As mentioned earlier, there needs to be communication between a NODE process and the HOST process in the Resource Manager Controller of the iPSC/2 system. At the beginning of program execution, the HOST process sends a message of GRAPH type bearing the program graph to all NODE processes in participating nodes. The key role of the PARSE engine is to process the message and build an internal data structure for the program graph by parsing the message. When the PARSE engine receives a packet of GRAPH type for the HOST process, it extracts the data part, i.e., the graph, and then parses graphs, while building an appropriate data structure for the graph. As depicted in Fig. 5, the data-structure consists of two parts : a program and a token structure. The program structure called "Fetch Hash Table" is provided to maintain the structure of input data-flow program graphs, while a token structure called "Token Queue" is provided for the initial tokens specified in the program. Since only the tokens that follow the mapping function for the tag value are enqueued in the Token Queue, initial tokens are distributed on the appropriate nodes[8][9].

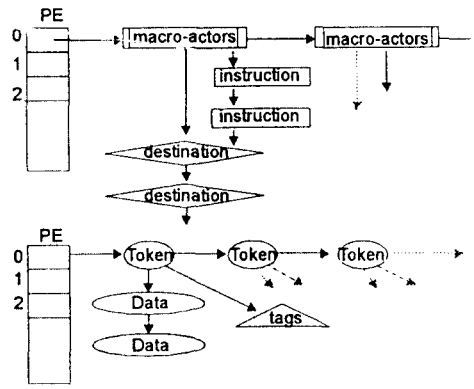

Fig 5. Data structure from Program graph after parsing

4.2.2 The MATCH engine

In the MATTDA machine, the number of input/output tokens for an actor is limited up to 5, respectively. The tokens consist of two parts : data and a tag. The tag field consisting of a collection of components, i.e., code block, statement and iteration field, is provided for matching purposes. The key role of the MATCH engine is to perform the matching operation for the input tokens and to create an activity to enable actors. The data-structures operated by the MATCH engine consist of a Token Queue(TQ), a Matching Hash Table(MHT) and a Enable Queue(EQ). The TQ is a first in first-out(FIFO) queue for storing the incoming token in the PE. The MHT shown in Fig. 6 is a hash table, each entry of which is a pointer to a linked list for storing tokens until the firing of the actor for the tokens. An entry of the MHT may hold tokens with different tag values, since the hash function is applied

only to part of tag components. The Enabled Queue is a FIFO queue for storing activities.

The MATCH engine consists of two parts : a communication service routine and a matching routine. When a message of the TOKEN type arrives at a PE, an asynchronous interruption in the MATCH engine occurs for the message. The routine extracts a token from the message and then prepares for the matching by enqueuing the token in the Token Queue. For the matching of incoming tokens, the MATCH engine dequeues a token from the TQ and then determines an entry in the HQ by calculating the hash function for the tag of the token. Starting from a first token linked to the entry, the HATCH engine attempts to match by comparing the tag of the token with that of the incoming token. If either no tokens are matched(Fig 7.1) or the incoming token is not the last token for a set of tokens for an actor while the matching succeeds(Fig 7.2), the token is enqueued in the hash entry(note here the storing location is different in either case). Otherwise(Fig 7.3), the engine extracts the remaining tokens for the hash table and forms an activity for an actor. The activity is then enqueued in the Enable Queue.

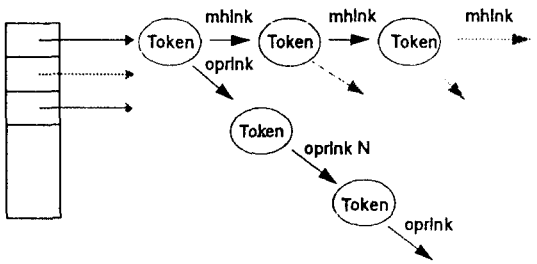

Fig 6. Match hash table

4.2.3 FETCH engine

In the MATTDA machine, an actor is a collection of instructions, i.e., a macro actor. On the firing of an actor, the ALU sequentially executes the instructions of an actor in a control-flow fashion of the von Neumann computer. The role of the FETCH engine is to emulate the operations of the FETCH hardware. The Fetch Hash Table (FHT) shown in Fig. 8 is a main data structure in the engine. Each entry in the FHT forms a queue for the instructions of the corresponding actors. The hash function of the FHT to an actor is composed of a linear combination of the identifier of the code block to which the actor belongs and statement number of the actor. The FETCH engine schedules an actor by dequeing an activity from the data structure storing activities. It

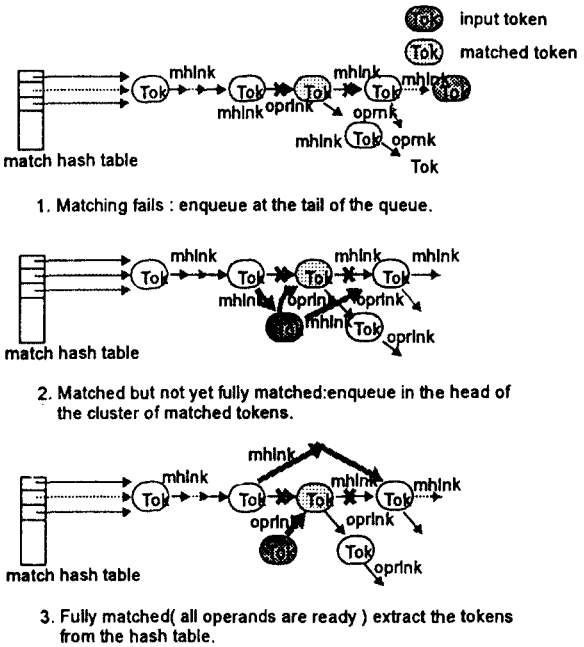

Fig 7. The operation for matching

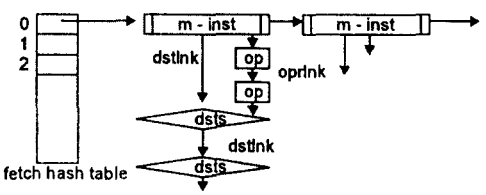

Fig 8. The structure of FETCH hash table

locates the code for the actor in the FHT by using the tag of tokens in the activity and setting a pointer with the location. The structure of the FETCH hash table is shown in Fig. 8.

4.2.4 The I-structure engine

The MATTDA machine has a global heap, I-structure for storing structure data. I-structure is a memory structure that has write-once semantic and employs split phase operations. The write-once semantics of the I-structure supports the fine-grain synchronization at a data level, while the split operation provides thesystem with potential to hide memory latency which is one of the critical problems of a multiprocessor system. In the current implementation, the I-structure is utilized to handle only the array type data structure[8]. Thus, cells in I-structure can only be accessed through array handling instruction. A more in-dept explanation of the mechanism and operational principles of the I-structures is as be low :

· Split-Phase Operation : For a READ request to an empty cell, the request is queued in the deferred list and is pending until the cell is written. When a WRITE command is requested on a cell with deferred requests, the requests are responded with the data carried in the command and the WRITE operation is performed on the cell.

· Centralized I structure : I-structure pools reside in a node which executes all array handling instructions generated in the system. The node works in such a fashion as a dedicated I-structure controller. This centralized I structure is simple in managing the I-structure since the name space for the array can be continuous and has a great deal of extensibility in implementation of the I-structure. In scientific application, the data structures are heavily accessed, which causes the centralized implementation of the I-structure to be a bottleneck of system performances.

· Distributed I-structure : I-structure are evenly

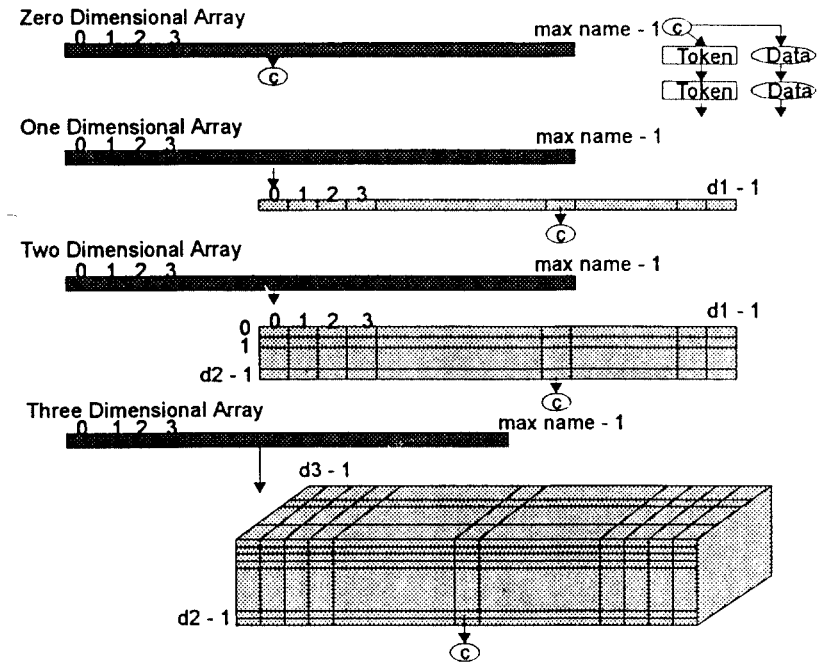

Fig 9. Internal representation of an I structure

distributed across the nodes in the system. All nodes in the system perform the service for the structure handling. This approach achieves the load-balancing of works related to structure handlings. The name for an array produced in a node must be consistent and be acknowledged in other nodes, which forms a global name space. In implementation, the names in a local node continually increase in numbers differing from corresponding global names. The global name for an array is assigned to maintain the information of the node in which it is allocated and the local name. In current implementation, the global name is the summation of the node id and value obtained by multiplying the local name with the count of nodes in the allocated the given program.

4.2.5 ALU engine and OUTPUT engine

An actor in MATTDA consist of a collection of instruction. When an actor is scheduled on ALU, instruction in the astor are sequentially executed in a control-flow fashion. The intermediate rsults of an instruction is transferred to the next instruction through register files. When the EXT instruction is encountered, the execution for the actor terminates. The MATTDA machine has five hundred register files. Each register can be addressed directly and indirectly in instructions. The register files are divided into five physical groups. Each group, which consists of one hundred registers indexed from zero to ninety-nine, corresponds to an input or output port. For example, the first group stores the token values of port 0. The major data structures used in implementation of an ALU engine are the Register File Butter and the Result Token Queue. The Register File Buffer is a data structure corresponding to the register file in hardware and has the same role in execution. The Result Token Queue is a buffer which stores the result tokens on ports. The Instruction set defined for the MATTDA machine is divided into six groups.

During the execution of an actor, instructions for token formation build data for tokens on the appropriate entry in the Output Token Buffer. The key function of the OUTPUT engine is to form output tokens and send them to the corresponding nodes. Using the Result Token Buffer in the list of destination actors in an activity, the Output engine forms the tokens by combining the data with tags. More precisely, for each destination node determined by using the mapping function for the tag of the token, a token is formed by duplicating the data value for the token which is stored in the Result Token Buffer, and the tag value of the destination actor attached to the tag field of the newly generated token. The complete token is temporarily stored in the Output Token Buffer. At the stage of token routing, if the destination node of a token is the same node as the current node, the token is directly enqueued at the tail of the Token Queue in the node ; otherwise, the token is packed into a message of TOKEN type and is routed to the destination node through interconnection networks.

4.3 Function Linkage Mechanism

In conventional von Neumann computers, an activation frame is provided to function invocation. The operations included in function invocation, i. e, parameter passing and result returning, are performed by suing the activation frame. In other words, the activation frame becomes the name of an instance of a function and the name space is defined as the memory range available for activation frames.

A function in data-flow graphs is represented by a re-entrant code-block specified with an input/output arc and a collection of actors. The formal input arguments to a function are tokens flowing into the function body. The formal output of a function is tokens produced out of the function body. Inside a code block, and actor is identified by a tuple (code block number, statement number). The code block number is the identifier for code block, and the statement number is the identifier of an actor. The tuple can be assigned statically at compile time.

In a data-flow computation, a tag is extended to name the instances of a function. A context field in the tag contains a unique value defined dynamic ally for the instance. Together with the static components of the tag, the context field constitutes the naming convention. The dynamic property of the context results in the intervention of the system resource manager. Whenever a function is invoked, the manager provides a unique context value for the invocation. After the fiction has finished, the context value is returned to the system resource manager for reuse.

4.3.1 Naming Convention

As mentioned earlier, a system resource manager, we call it context manager, is required to service the naming of dynamic contexts. Practically, there are two schemes in managing contexts: centralized and distributed management. In a centralized management, there is a only single context manager in a system, whereas one con text manager exists on every PE in distributed management. Even though a centralized manage ment is simpler than a distributed case, it suffers from the low scalability in that the centralized manage may be a bottleneck when the number of PEs are larger.

In current implementation, we choose distributed management to balance the load evenly on every PE ; Every PE has a context manager which generates a globally unique name for a function invoked at the PE. The size of context allowed to each context manager is a hundred. Thus, the maximum number of functions invoked simul taneous in a system becomes is PE*100.

4.3.2 Argument Passing and Result Returning

Since arguments and results of a function in a dataflow are passed to a function body in the form of a token, the key operations are related to changing the tag, particularly the context field of the tag. In the argument passing, the context component in the tag of the argument token is replaced with the context of the callee's obtained from the context manager. On the other hand, for result returning, the caller's context extracted from an argument token during argument passing is restored at the context field of result tokens.

4.3.3 Actors for Function Linkage

Special actors are provided to support the func tion calling. The actors are GXT, ETG, CTG and RTX : The GXT actor provides a new tag for a function. The ETG actor extracts a tag value from an input token[5],[10]. The CTG replaces the context part of the tag of a token on one input with that provided on the other port. The RXT actor replaces the tag of an input token and returns the context to the system resource manager.

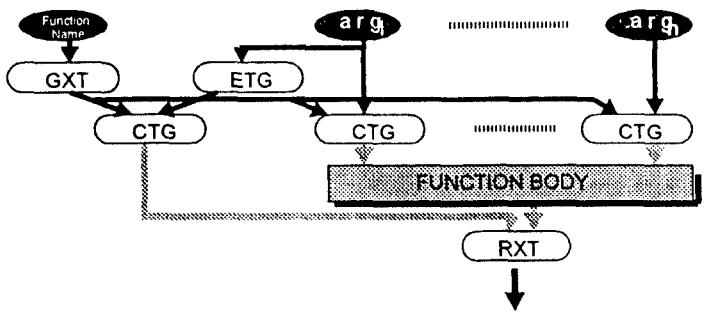

Fig 10. Overview of function calling mechanism

· GXT actor : a resource management actor, the GXT obtains a context from the context manager and provides a new tag for a function by combining the code-block and a context. The input of he actor is a code block number for the function. The output token of the actor carries a tag value formed by the combination of a context and the supplied code block number.

· ETG actor : the key role of the ETG actor is to prepare the result returning by providing the required information. Two kinds of input are supplied to the ETG actor : an argument token and the destination information. Given the input argument token, the ETG actor extracts the caller's information such as iteration field, the caller's context number and code block number. On the other hand, the destination information is supplied as operands of the actor. The information the includes statement number, token count to fire and port number of the destination actor. Using this information, the ETG actor generates a tag for an actor to which the function result will be passed.

· CTG actor : The input of the CTG actor is a token with a context number and code block number for the function on port 0 and a data token on port 1. Similar to the ETG actor, the information of destination such as the statement number, token count to fire and the port number are supplied as operands. The CTG actor replaces the context, code block number and statement number components of the argument token's tag.

· RXT actor ; The inputs of the RXT actor consists of two tokens bearing a destination tag and returning data, respectively. Like the GXT actor, the RXT actor is also a system management actor, performing two operation ; First, the RXT actor returns the context of the function to the context manager for reuse. Next, the actor forms a result token by combining the tag and returning the data supplied as input.

V. Performace evaluation

5.1 Algorihm Description

5.1.1 Floyd's Algorithm

Floyd's algorithm is for a method calculating the distance of shortest paths for all pair of nodes in the directed graph G. It is based on a heuristic to divide the distance into two logical groups and to select the smaller one as the local minimum [11],[12]. For $\mathbf{node}_k$, the distances from $\mathbf{node}_i$ and $\mathbf{node}_j$ are two cases : one is the distance when $\mathbf{node}_k$ is considered as a via **node** and the other is when $\mathbf{node}_k$ is not considered. In this case, the smaller one of the two is the candidate for the shortest distance between the two nodes since the two cases of distances are distances for distances paths from $\mathbf{node}_i$ to $\mathbf{node}_j$.

By generalization, the shortest distance can be obtained by iterative applications of nodes and choosing the smaller one as the new local minimum distance between the minimum distance obtained through proevious iteration and the distance newly obtained in the iteration.

The algorithm is expressed by two classes of iteration. One class is the iteration for applying via nodes and the other class is for generating all pairs of nodes in the digraph. In the algorithm, the digraph G is represented in adjacent matrix a which's two-dimensional. The element A[i][j] is the distance between $\mathbf{node}_i$ and $\mathbf{node}_j$. The rough desription of the algorithm is outlined below :

```
Algorithm Floyd (IN N : int, A : array OUT
                 B : array)
  begin
   for k = 0 to N do
      for i = 0 to N do
         for j = 0 to N do
            begin
               if A[i][k] + A[k][j] < A[i][j]
                  B[i][j] = Aik + Akj ;
            end
  end ;
```

5.1.2 The LU decomposition

The problem of **LU** decomposition was formally depicted solving **L** and **U** for a given matrix **A** where **A**=**LU** and **L** is *lower triangular* and **U** is *upper triangular*.

For the case of a 4*4 matrix,

$$\begin{bmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{bmatrix} = \begin{bmatrix} \alpha11 & \alpha12 & \alpha13 & \alpha14 \\ \alpha21 & \alpha22 & \alpha23 & \alpha24 \\ \alpha31 & \alpha32 & \alpha33 & \alpha34 \\ \alpha41 & \alpha42 & \alpha43 & \alpha44 \end{bmatrix} \begin{bmatrix} \beta11 & \beta12 & \beta13 & \beta14 \\ \beta21 & \beta22 & \beta23 & \beta24 \\ \beta31 & \beta32 & \beta33 & \beta34 \\ \beta41 & \beta42 & \beta43 & \beta44 \end{bmatrix}$$

The i, j^{th} components of the above equation are represented in three cases.

$$i<j \Rightarrow aij = \alpha i1\ \beta 1j + \alpha i2\ \beta 2j + \ldots + \alpha ii\ \beta ij$$
$$i=j \Rightarrow aij = \alpha i1\ \beta 1j + \alpha i2\ \beta 2j + \ldots + \alpha ii\ \beta ij$$
$$i<j \Rightarrow aij = \alpha i1\ \beta 1j + \alpha i2\ \beta 2j + \ldots + \alpha ii\ \beta ij$$

The Crout's algorithm quite trivially solves the set of N2+N equations for all the α's and β's by just arranging the equations in a certain order [13],[14]. That order is follows :

· Set $\alpha 11 = 1$ and $\beta 11 = a11$

· For each j = 2,3,...,N do these two procedures : for i = 2,3,...,N solve for β_{ij} according to the following equation.

$$\beta ij = \alpha ij - \sum_{k=1}^{i-1} \alpha ik\ \beta kj$$

Next, for j=2,3,...,N solve α_{ij} in the equation :

$$\alpha ij = \frac{1}{\beta jj}\ \alpha ij - \sum_{k=1}^{i-1} \alpha ik\ \beta kj$$

5.1.3 Householder Algorithm

A matrix B is upper Hessenberg if $bij = 0\ i>j+1$. householder algorithm is a direct and fairly efficient way to reduce any square matrix to upper Hessenberg by unitary similarity transformation.

〈The principle of the algorithm.〉

Let x be any nonzero vector with real coefficients.

Define

· $\sigma = \pm \|X\|_2 \sigma$

· $u = X + \sigma e_1$

· $\pi = 1/2\|u\|_2^2$ ·

Then $R = i - (1/\pi)uu^H$, is a Householder transformation[14],[15], is a unitary matrix with the property that $Rx = -\sigma e$

〈Description of the algorithm.〉

The following is the description of k_{th} iteration where k = 0 to n-2. On each k_{th} step, $A_{kj} = 0$ for $k < j-1$

· Takes a vectpr($x = a^{k-1}_{k,k-1}, a^{k-1}_{k+1,k-1}, \ldots\ldots, a^{k-1}_{n,k-1}$) for matrix A_k

· Computes the Householder transformation for R_{k-1} for the vector X.

· Takes U_{k-1} to be the matrix

$$\begin{bmatrix} ik-1 & 0 \\ 0 & Rk-1 \end{bmatrix}$$

· Takes $U^H{}_{k-1}$ to be matrix

$$\begin{bmatrix} ik-1 & 0 \\ 0 & R^H k-1 \end{bmatrix}$$

· Calculate $A_k = U_{k-1} A_{k-1} U^H{}_{k-1}$.

5.1.4 The QR Algorithm

The QR algorithm is widely considered to be the eigen problem method of choice for all matrices except sparse matrices of large order[16],[17].

〈The principle of the algorithm.〉

Let A be any square matrix for which the eigen value is to be computed.

· Factorizes A as followings

$A - \alpha I = QR$

Compute new matrix A

new $A = RQ + \alpha I$

After substituting this relation for R_k, we have

new $A = QAQ^H$

· If we iterate the previous two step, we get a sequence of **A**'s that converge to a matrix $\bar{A}$ of the form

where θ is approximation of eigen value. The QR algorithm process stops at A_N when the coordinates $(x = a^N_{n,1}, a^N_{n,2}, \ldots, a^N_{n,n-1})$ of A_N are negligible.

$$\bar{A} = \begin{bmatrix} C & h \\ 0^T & \theta\bar{k} \end{bmatrix}$$

〈Description of the algorithm〉

The QR algorithm with origin shift consists of construction of the iterative process of matrices A_k according to the following recursive rule :

· $A_0 = A$

· For $k = 0, 1 \ldots, N$ compute matrices Q_k and R_k with Q_k unitary and R_k upper triangular, which satisfy the factorization

$$A_k - \alpha I = Q_k R$$

where $\mathbf{R} = \mathbf{S}_{n,1} \ldots, \mathbf{S}_{3,2}, \mathbf{S}_{2,1} = \mathbf{Q}_H \mathbf{A}$ and S_{pq} is plane rotation matrix and α_k is the Rayleigh quotient of the vector e_n applied to A^H_k.

· Compute

$$A_{k+1} = R_k Q_k + \alpha I$$

5.2 Performance of Algorithm

We implemented the above algorithms using data-flow graphs and parallel-C programs, respectively. For example, the data-flow graph for the Floyd algorithm is shown in Fig. 12. The data-flow graphs are executed on the emulator, whereas parallel-C programs are executed on iPSC/2 without the emulate layer. For performance evaluation, the execution time of the algorithms are measured, and corresponding speed-ups calculated. They are depicted in Fig. 11 and Fig. 12 for each case.

Overall, the two approaches do not show the ideal speed-up, i.e., the speed-up proportional to the processor number. The main reason is that the interconnection network speed of the iPSC/2 system is extremely slow. The iPSC/2 system, a distributed memory parallel system, provides a facility to communicate via message exchange among PEs. Compared with other parallel machines available, iPSC/2 system has a relatively slow interprocessor communication time as shown in Fig. 13 where io_{oh} is the fixed overhead to construct a message and set up the appropriate communication mechanism, io_{dp} is the time to send a double precision number. Consequently, the low communication speed causes low system utilization as both the data-flow and parallel-C implementation spend a large of time on communication.

As shown in Fig. 12 and 13, the data-flow approach is better in its performance under the scalability perspective. The main reason for the result is that the dynamic scheduling of the data-flow approach is more suited to exploit the

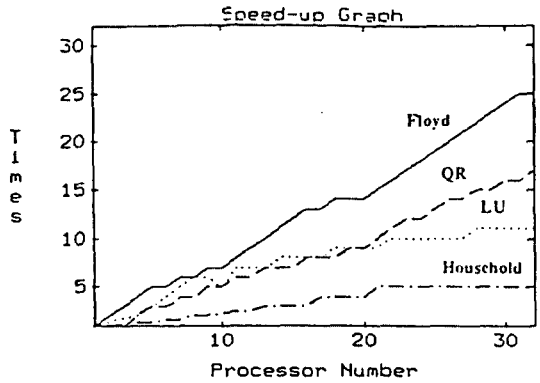

Fig 11. Speed-up on the emulator(data-flow graph)

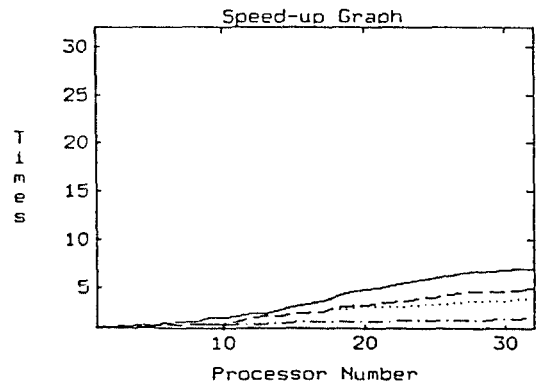

Fig 12. Speed-up on the iPSC/2 parallel-C implementation

Model	iooh(us)	iodp(us/dbl)
NCUBE	384	10.4
iPSC/2SX	900	3.4
iPSC/2VX	900	3.0
Wrap	0	0.4
iWrap	0	0.2
T800 - 20	0.95	4.6

Fig 13. Communication and computation parameter for various computer

fine-grain parallelism. On the other hand, from the parallel-C implementation, we observe that algorithms with irregular structures are very difficult to parallelize since users have to consider the synchronization among multiple concurrent tasks. Thus, parallelism in the algorithm can not be efficiently exploited, resulting in low system utilization. During synchronization, a PE may be idle without computing nor actively communicating with one of its neighbors. When we define the synchronization overhead as idle time(t_{idle}), we can formulate t_{idle} of an algorithm mapped on a p processor as

$$t_{idle} = t - \frac{\sum_{i=1}^{p} (t_c^{(i)} + t_{io}^{(i)})}{p}$$

where t is the execution time, t^i_{io} is the time that the i^{th} PE spends actively communicating with its heighbors, and t^i_c is the time that the i^{th} PE spends doing local computation. In the parallel-C implementation, the large idle time caused from the synchronization over results in the poor-up shown in Fig. 13. In summary, the data-flow approach is better than the conventional parallel approach both in programmability and parallelism exploitation.

VI. Conclusion

In this paper, we present the experimental results of data-flow computation versus conventional parallel programming approaches on a von Neumann parallel machine. We design and implement a data-flow emulator on the iPSC/2 system. Using the emulator, we experimented with data-flow computation by implementing a number of algorithms in data-flow graphs. The performance of the algorithms in the data flow appoach is compared with the cunterpart of the conventional parallel programming approach.

Form experimentation, we show that the data-flow approach is better conventional parallel programming. In the iPSC/2 system, the user has to partition and allocate the program. It is extremely difficult to design an efficient parallel algorithm since the user has to specify all the synchronization names and operations explicitly through communication primitives according to criteria of optimally reducing the frequency of synchronization and the total idle time which are hardly estimated in programming time. Overall, the synchronization proved to be large grain and caused low system utilization, while adding difficulty in the parallelization of algorithms.

In a data-fow machine, sysnchronization is supported in the hardware, which implies that the synchronization is implicit. Regardless of the number of PEs, program behavior is deterministic. Due to the implicit synchronization and deterministic behavior of a program, the problem related to synchronization is transparent to users. Therefore, the degree of programmability on the dataflow system is comparable to that on a single processor system. However, we can not avoid the overthead of the emulation based on software. In order to achieve a more realistic performance evaluation, more elaborated emulation based on a high rate of hardware components or a practical data-flow machine is desired in the future.

Acknowledgement

Y.D. Lee wishes to thank the Ministry of Education for the financial support. This paper is based on the work performed at University of

Southern California while he was visiting Professor. Also the author would like to express his appreciation to Professor J-H Gaudiot and his Data-flow group members at U.S.C. for their supports of this research.

본 논문은 1991년도 교육부 학술연구조성비에 의하여 연구되었음.

References

1. Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessors : the data-flow solutions." Technical Report LCS/TM-241, Lab. Comput. Sci., M.I.T., Sept. 1983.
2. J. L. Gaudiot, "Data-driven multicomputers in Digital Signal Processing," Processings of the IEEE, vol. 75, no. 9, sept. 1987.
3. J. L. Gaudiot and Yi-Hsiu Wei, "Token Relabeling in a Tagged Token Data-Flow Architecture," IEEE Transactions on Computers, vol. 38, no. 9, Sept. 1989.
4. J. L. Gaudiot and Namhoon Yoo, "Marcro Data-Flow Simulator," Department of Electrical Engineering-Systems, USC, Los Angeles, California, Oct. 1990.
5. J. L. Gaudiot and M. Ayed, "Data-Flow Assembly Language," Department of Electrical Engineering-Systems, USC, Los Angeles, Califonia, Nov. 1990.
6. Robert A. Iannucci, "Toward a Data-Flow/von Neumann Hybrid Architecture," Proc. of the 15th Annual International Symposium on Computer Architecture, pages 131-140, 1988.
7. Intel Cop, "iPSC/2 Simulator Maunal," Tech. Memorandum 143, Argonne National Lab., Nov., 1990.
8. Arvind and R. E. thomas, "I-sturctures : An efficient data type for functional languages," Tech. Rep. LCS/TM-178, Lab. Comput. Sic., M.I.T., june 1980.
9. Arvin and K. P. Gostelow, "The U-Interpreter," IEEE Computer, pages 42-29, Feb. 1982.
10. Guang R. Gao, "A Code Mapping Scheme for Data-Flow Software Piplining," Kluwer Academic Publishers.
11. L. Fox, "An Introduction to Numerical Linear Algebra," Oxford University Press, New York, 1965.
12. N. Gastinel, "Analyse numerique lineaire," Hermann, Paris, 1966.
13. Alton S. Householder, "The Theory of Matrices in Numerical Analysis," Blaisdell Publishing Company.
14. Alto S. Householder, "Principles of Numerical Analysis," Dover Publications, New York, 1953.
15. Alton S. Householder and F. L. Bauer, "On certain iterative methods for solving linear systems," Num. Math. 2, 55-59, 1960.
16. W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, "Numerical Recipes, The Art of scientific Computing," Cambridge University Press, 1986.
17. F. Szidarovszky and S. Yakowotz, "Principles and Procedures of Numerical Analysis," Plenum Press, New York, 1978.

李 龍 斗(Yong Doo Lee) 正會員

1952년 5월 15일생

1975년 2월 : 한국항공대학 통신공학과 졸업

1983년 2월 : 영남대학교 대학원 전자공학 석사

1991년 9월 : 한국항공대학 대학원 전자공학 박사과정수료

1981년 8월 ~ 1982년 2월 : 日동경대학생산기술연구소 객원연구원

1991년 8월 ~ 1993년 2월 : 美Univ. of Southern California 객원교수

1982년 3월 ~ 현재 : 대구대학교 공과대학 전자공학과 교수

※주관심분야 : 컴퓨터구조, 컴퓨터네트워크

蔡 洙 煥(Soo Hoan Chae) 正會員

1950年 10月 28日生

1973年 2月 : 韓國航空大學 電子工學科 卒業(工學士)

1973年 7月 ~ 1977年 8月 : 空軍技術將校

1977年 8月 ~ 1983年 7月 : 金星通信研究所

1983年 8月 ~ 1985年 5月 : 美國알라바마 大學電算學科(理學碩士)

1985年 5月 ~ 1988年 12月 : 美國알라바마 大學電氣工學科(工學博士)(電子計算機構造專攻)

1989年 3月 ~ 現在 : 韓國航空大學 電子計算學科 助敎授