Systems and Computers in Japan, Vol. 24, No. 10, 1993
Translated from Denshi Joho Tsushin Gakkai Ronbunshi, Vol. J75-D-I , No. 11, November 1992, pp. 1025-1036

Design and Implementation of the C// Language for Closely Coupled Parallel Computers That Control Distributed Processes by Demand/Accept Mechanism

Masaki Tomisawa, *Member*, Hitoshi Tamura, Hiroshi Hoshino, *Nonmembers*,
Satoshi Igarashi, Oichi Atoda and Nobuo Saito, *Members*

Faculty of Technology, Tokyo University of Agriculture and Technology, Konganei, Japan 184

SUMMARY

Design and implementation of the parallel programming language C// is discussed. C// is so designed as to give straightforward programs to shared memory parallel computers equipped with a distributed "demand/accept" control mechanism previously proposed by the authors, while offering moderate abstraction and descriptive power as a complete programming language as well as friendliness to programmers. The language itself is made as light as possible for easy implementation. A function of the language C is interpreted as a parallel grain in C//, introducing the "advance call" as the only parallel fork method. New concepts of multiplicity and sharedness are added to value and address passing in sequential case. Eight parallel primitives are defined for manipulating control advance flow as well as data passing. The structure of a C// program consists basically of nested advance calls spreading in both width and depth. A program is written on the basis of logical parallelism without specification of processor element usage. Logical "function instances" are distrubuted to physical processor elements at runtime and executed as native processes that the control mechanism manages. The optimal scheduling strategy is left open. The language is empirically evaluated through a picture analysis program.

Key words: C// parallel programming language, shared-memory parallel computing, distributed "demand-accept" control mechanism.

1. Introduction

From the point of view of machine as well as application of parallel computers, it can be said that SIMD computers have been quite successful in pracitcal use. On the other hand, many independent parallel programming languages [2] have failed to bring MIMD parallel computers [1] into practical use. By nature, the role of a programming language is to overcome the gap between machine architecture and user applications. Introduction of machine oriented, though not machine dependent, parallel programming languages [3, 4, 5] and language oriented parallel architectures [6, 7, 8] is essential for mediation between MIMD machines and actual application programs. Transputer [9] is one of the few successful examples in this viewpoint.

A programming language for this purpose should be so designed as to properly abstract the resources and operations of target machines, to give efficient machine codes, and to be compiled with no difficulty. On the other hand, in view of the original role of a programming language, it should offer proper programming paradigm and good programmers' interface.

In this paper we present C//, a parallel programming language appropriate for MIMD computers whose architecture, which was proposed by the authors before, is characterized by a *demand/accept* control mechanism [10] built into all processor elements distributely. Expression of parallelism in C// presupposes dynamic generation and extinction of processes done by that control mechanism. Linguistic mod-

ISSN0882-1666/93/0010-0015

eling of control flow or synchronization in C// is not novel and can be easily learned.

Differnent from the C-thread in Mach operating system [11], parallel primitives of C// are not system library functions. The operations of the demand/accept control mechanism are abstracted as native operators within language syntax as well as jump-to-subroutine in a serial machine is abstracted as the function call syntax of C. This comes from the basic concept of C// to support machine oriented context of user programs. Though the conceptual bases are differnet, C// and C-thread share comparable features concerning dynamically generated processes (the term does not mean Mach proceses here).

C// functions are executed directly in parallel without the vehicle jof threads. In Mach, a thread generated by cthread_fork() need not be joined by cthread_join(). Instead, it may be thrown away by cthread_detach(). In C//, a called function instance is subject to the caller context and need be joined at the reception operator // attended with a return value (including void). This is appropiate for the distributed strategy of mutual process control by demand/accept mechanism. Whenever a C// function is invoked in parallel with the caller, a function instance is generated and is bound to a *transfer variable* explicitly. The actual content of a transfre variable is an internal identifier of the function instance which is hidden from the program contest because it is hardware dependent. Instead, the declared name of a transfer variable serves as the temporal name of a function instance independent of the function name and makes the programmer conscious of that function instance. There is no syntactic distinction among a quasi-SIMD program in which many function instances with different parameters are made from a function, a recursive program in which function instances are diffusively created, and a program in functional distribution in which many different functions generate small number of function instances for each. Only three essential function call primitives respond to a wide range of parallel program structures, which, not a variety jof separate primitives, makes C// a general purpose parallel programming language. Similarly in C// in only one lock-statement is used in place of the class of mutex_try_lock(), etc.

In this paper we do not mean to propose a programming language based on a new theoretical parallel model. The purpose of this paper is to show a design of practical parallel programming language which makes MIMD computers with demand/accept control mechanism widely usable.

2. Machine Architecture and Application Programs

A closely coupled MIMD parallel computer having the following features is assumed in this paper.

(i) Processes are distributively created and deleted by the efficient control mechanism built into each processor element which is manipulated through machine level instructions. Each processor element executes a number of processes concurrently. The control mechanism is responsible for context switching among concurrent processes and for prevention of processor starvation. The environment of a process is a heap allocated by the control mechanism within the home memory of the processor element. The control mechanism operates roughly as follows. Mnemonic codes of the instructions are represented by capital letters. A certain process executes DEMAND instruction with data pointer grain from a target processor element. By DEMAND instruction, the control mechanism of demandant processor element fills slots of a data structure named *recipient* in the *instance control block* within the home memory of that processor element and puts the recipient address into the queue of the target processor element. The control mechanism of the target processor allocates a new heap and creates a process in demand. When the created process executes DELIVER instruction, the heap is released and the process is deleted. Returned data may be delivered to the environment of the demandant process before DELIVER instruction. ACCEPT instruction of the demandant invalidates the recipient if the target has DELIVERed. If not, if waits switch-spinning until it has done so. Consequently, the demandant process can accept the delivered data after ACCEPT instruction. Instructions for selection of an arbitrary process among processes having DELIVERed and forced termination of a process are also implemented in the demand/accept control mechanism. Parallel control instructions for the mechanism, which are presented in [12] and revised afterwards, are listed in Table 1. Owing to these instructions, the parallel computer executes a parallel program autonomously without the aid of an operating system.

(ii) Address space is logically linear and uniform, all of which is shared by all processors. Physically, memory space is divided into as many blocks as processors, each of which is called a home memory and mounted on a circuit board with a processor chip and an arbitration circuit to form a processor element. A processor element can access its own home memory at the lowest cost. It takes 1.2 to 5 times longer time,

Table 1

Instruction	Input operand	Output operand	Function
DMND	Start address and parameter address	Recipient address	Demand (parallel call)
ACPT	Recipient address	Parameter address	Acceptance (with switching)
DLVR			Delivery (return of demand)
SLCT	First recipient address in queue and number of elements	Element number	OR-type selection of reception point for delivery (with switching)
RTCT	Recipient address		Cancellation of demand
SETR			Sequential registration of domain boundaries
RMVR			Sequential demand cancellation in domains
TASS	First address of test and set data address queue and number of elements		Atomic test and set of all elements in queue and switch spinning
AOF	Immediate mash comparison value		Freeze, exception (for debugging)

which may not be uniform and may be affected by bus confliction, for a processor to access a home memory in another processor element through the bus interconnection network.

(iii) The number of processor elements, physical bus scheme, bus interconnection geometry, sequential instruction set, presence of cache memory and so on are not prescribed.

Structures of application programs that the parallel computer is assumed to execute are as follows.

(A) A program which divides a set of large uniform data such as pixels into many subparts at a time and processes them in parallel. In some cases SIMD computers can be used for this purpose, but, in other cases in which respective subcontexts in parallel take different branch conditions or different processing status, MIMD computers are appropiate. In picture processing examples, convoluting a filter grid by grid is for the former, and tracing edges polyhedron by polyhedron in various shapes is for the latter.

(B) A program in which state transitions take place and the context forks into as many subcontexts as states reachable from the present state. In general, the subcontexts fork repeatedly, in many cases recursively, to become tree-structured. Breadth first exploration of a search tree is a common example.

(C) A program in which respective subparts in parallel work differently.

In (A) and (B) among the above, high degree of parallelism can be attained with a small amount of program description. For (C), we have not encouncountered highly parallel practical problems, but the nested structure of (C) in large problems jsuch as structural pattern recognition and so on may produce good parallelism as a whole. Compound structures of (A), (B) and (C) are possible.

3. Criteria and Method of Language Design

3.1. Requirements for language

Regarding language design, there are many requirements for target processor, compiler, programmer, etc. The following are considered.

First, a processor element that executes a part of the parallel program must be explicitly demended for its process creation. The compiler must generate executable codes on the basis of explicit DEMAND instruction. That is, the following is necessary.

Model. That a process will be created must be known before the process is created. In short, control-driven code is obtained.

For efficient execution of the code, the following two are required.

Local Access. Import or export of data should be done at the time of creation or extinction of a process as parameters or return values. Locked reference to data of other process which is possibly in the home memory of other processor element should be infrequent.

Optimal scheduling. Optimal assignment of processes to processor elements is obtained which may significantly affect the total execution time.

To write a compiler of the language easily, the following is required.

Semantic resemblance. Data model and control flow model of the language should not be too far apart from the memory model and control model of the hardware for the compiler to absorb the semantic gap between them.

The following is desired for both easy compilation and runtime efficiency.

Code simplicity. Code should be as light as possible without heavy buried macros or system calls. Codes with many dynamic checks of type or area should be avoided. Implicit allocation or release of a heap aside from those done by demand/accept control mechanism may degrade execution speed seriously.

In general, the requirements from a programmer are the following two items.

Simple programming. Easily learnable syntax, fluent coding without error, effortless debugging and so on are desired by every programmer.

Descriptive power. The wide range of programmer's thought must be expressed by language primitives without unnatural or detoured tricks. However, above two often conflict when a number of various primitives are introduced in favor of the latter.

Finally, a programming language must have the following features in the least.

Completeness. A programming language must be complete under the paradigm which it is going to offer even if it is a machine-oriented one.

Portability. A programming language must cope with hardware variation. In our case, the number of processor elements and interconnection geometry may vary from machine to machine.

3.2. Establishment of design criteria

For each requirement, the criteria for trade-offs are established as follows,

(I) To attain both **semantic resemblance** and **simple programming** in order for a programmer to program plainly making an effective use of demand/ accept control straightforwardly.

(II) To support applications in the range of (A) to (C) sufficiently, but not to expand faculty in preparation for unexpected usage. **Simple programming** and **descriptive power** are balanced at this point, keeping the language model and specifications as plain as possible.

(III) While making the language machine-oriented in the sense that is assumes the demand/accept control mechanism and the linear shared address space, to release it from the constraints of individual machines such as different processor element numbers or bus geometries. **Portability** is attained thus among parallel computers equipped with that control mechanism. Since **optimal scheduling** depends on the physical schemes and parameters of individual computers, it should be left outside the language syntax or semantics.

(IV) To make implementation of the language processor easy while satisfying (II) and (III). **Code simplicity** is regarded especially important.

(V) To borrow conventional programming language syntax as a subject and to make a smoothly continuing extension for parallel description, which contributes to **simple programming** since only the extended part of syntax must be learned by a programmer. In addition, **completeness** should be considered for the extended part only. This reduces the design effort to a large extent, and improves the implementability.

3.3. Design of the language structure

According to the trade-offs made in 3.2, a model of the language is determined first. Based on Criterion (I), a call type procedural parallel language is the most appropiate. Namely, a process is created when and only when a subprocedure is called and control is given to it explicitly, which satisfies the **model**. However, from (III), parallel description is made not subject to physical processor elements but on the basis of logical processes. Thus we have decided that the first machine oriented language for parallel computers with demand/ accept control be procedural language which abstracts the process creation on a logical level as invocation of subprocedure or C-like function. Here, as the first language of the demand/accept type computer, such a procedural call type language is assumed, and various criteria are satisfied within this framework.

A subprocedure might be called many times in parallel to take charge of respective parts of data in an

Table 2. C// parallel primitive series

Series	Parallel function	Name	C// primitive
Both series	Establishing control	Advance call	// = operator
			Transfer variable
Passing by value return series	Data multiplicity	Passing by value	(C)
		Automatic variable	(C)
	OR-type control selection		Among statement
	Synchronous termination of control	Return	(C)
		Receive	// operator
		Cancel	/! operator
	Cancellation of data multiplicity	Return value	(C)
		Delayed substitution	$ operator
Passing by reference pointer series	Sharing	Passing by reference	(C)
	Shared variable reference	Pointer reference	(C)
		Lock	Lock statement
			Keyhole variable
	Multiple representation	Struct	(C)

Note: (C) indicates use of corresponding C function

application of type (A). In (B), a subprocedure might be called in which reachable states are sought and which calls itself recursively as many times as the number or reachable states in parallel, giving data for respective states. Various subprocedures which work differently might be called at the same time in (C). Thus the difference of program structure between (A) and (B) is simply the depth of nested calls. (A) and (B) are different from (C) in that they call the same subprocedure many times simultaneously instead of many different subprocedures at a time. Then only one way of procedural fork, which is to call a subprocedure, will work for (A), (B) and (C) satisfying **descriptive power** in (II) if the condition below is met.

(1) A subprocedure call including recursive one can be expressed in the same syntax irrespective or multiplicity or depth.

There are search problems of type (B) which come to an end when at least one solution is found. Nondeterministic selection of a process which has been completed among specified callee processes is required to maintain **descriptive power** against those problems. Explicit expression of forced deletion of a callee process is also required. Another case is when nondeterministic selection plays its role to compose a program structure in which the main procedure decides what to call next whenever a subprocedure finishes. Anyhow, besides deterministic expression of procedural join, nondeterministic selection is inevitable in describing the parallel control flow.

Interaction of processes without fork or join of control flow can be done through a shared variable. For example, in a breadth first search program of type (B), branches are bound by competitive searching of a optimum value shared by all processes. In accessing a shared variable, mutual exclusion is of course indispensable. However, more complicated exclusion or synchronization is seldom encountered in applications of type (A) or (B). Even if complicated ones are needed, it is well known that they could be composed of the simplest exclusion logic. Thus a primitive for atomic lock sufficiently satisfies **descriptive power**. Composition of sophisticated synchronization and prevention of deadlock which are left to programmers will not degrade **simple programming** seriously if they do not appear too often.

The language leaves explicit description of control flow to the programmer's hand. There arise questions as to whether or not a programmer can manage the parallel control flow easily and whether or not a programmer can make up a parallel program structure without excessive effort. If these are not easily available **simple programming** in (I) and (II) cannot be satisfied. However, these are problems unique to parallel programming, and there are many open problems due to the lack of cumulative experiences. Our language copes with the former question by the following strategy.

(2) Control is made not to fork unless the control of a subprocedure is given an explicit name at the head, by the aid of which a programmer is able to seize

the corresponding tail of the control definitely. The name is declared like a variable before it is given to a control, which gives a chance for some static check.

For the latter, control flow is so organized that the classical program structuring method may be introduced extensively to parallel programming.

(3) Control that have forked away from a control must always join into that trunk unless it is terminated by force. The language syntax and semantics forbid detachment. Namely, only nested context is allowed.

Above are the basic strategies in designing the language. Some supplemental discussions are given below. The structure in (2) and (3) does not support programming models in which control is implicity bound to a variable when its value is referred to or when it can be evaluated. A finite number of callee processes can exist for a callee processes can exist for a caller process at once, which is not appropiate for the program model which requires theoretically infinite processes at once. Were it to meet requirements different strategies other than a call type procedural language would have been used from the beginning. In case of applications from (A) to (C), **descriptive power** is not too small since a finite number of data parts or reachable states are known before the call of subprocedures. Such a structure is rather favorable to (IV) since conventional compiling technique is applicable.

The control flow structure in (3) does not allow process creation by a process on the same nesting level as itself and direct data passing to a new process, which may be suitable for certain applications of type (A) when processings are done step buy step depending on the results of respective steps. Simple detection of an error in intermediate results could be written in this structure. In this case, direct transfer is not necessary a common caller which inspects the result of the foregoing process and indirectly call the following process depending on it is sufficient. Thus **descriptive power** is not affected seriously.

It is not so difficult to extend a conventional serial language which permits recursive calls to create a parallel programming language structure having features (1), (2) and (3). If we choose language C as the base, it is natural to adopt a function in place of a subprocedure and to create a new process when a function is called. The only thing to do is to introduce a way to express giving multiple controls to a function or giving controls to multiple functions simultaneously. Especially C is favourable to **semantic resemblance**

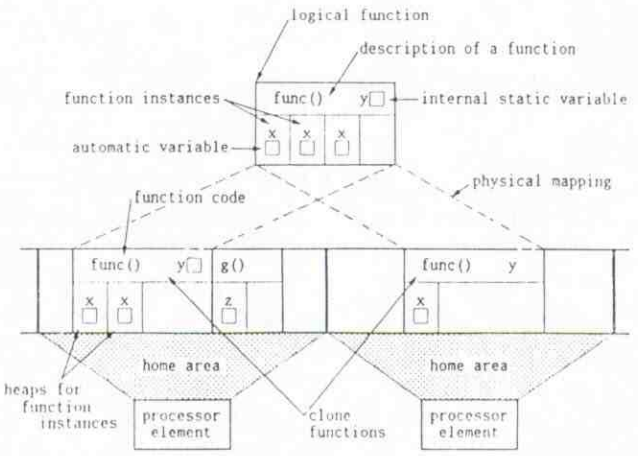

Fig. 1. Clone of function codes and function instances distributed to home memories of respective processor elements for mixed execution of concurrent and parallel ones.

since it processes machine-oriented descriptions. Thus we decided that our language is to be designed as a continuous superset of C. Compatibiltiy of system libraries functions out of the language syntex is not considered.

Giving the concept of scop to variables and accelerating the use of automatic variables as well as value passing reinforce **access locality**, increasing the chance of access to the home memory. However, it is very difficult to determine optimal assignment of processes which deal with nonlocal data in (A) and (B) and home memories considering not only load balancing of processor elements but also nonuniformity of access time or bus competition (conflict). Satisfying **optimal scheduling** automatically [13] is an open problem. If the location of a created process were to be specified in a program context by a human programmer, **portability** would be lost since the program would be dependent on hardware geometry. Our strategy is to leave this problem out of the design of language. Even if the spatial scheduling is nonoptimal, it is optimal for implementing parallel computers sooner.

There are some applications in (B) in which copies of large dat sets like prolog global stack must be passed from a state to many successive states because the data may be modified differently from state to state. Speed improvement of those programs is out of the scope of the language, and is left for programming sophistication or special purpose architecture [14].

4. Specification Design

4.1. Parallel grains

Language specifications are given so as to satisfy **completeness** while realizing the discussions given in 3.

As discussed, a subprocedure is to be expressed as C-like "function," which is essentially a procedure. In this language, a function call is the only way to fork control. A function call is a procedural call and a logically existing state as a process is referred to as a function instance. Of course there can be many function instances of a function simultaneously. Environment of a function instance is not a stack but a heap which the demand/accept control mechanism offers.

Function instance is a logical concept. Whether function instances are executed concurrently or in parallel physically is not questioned. This satisfies (III). In applications (A) and (B), many function instances of one function are made. In order for them to be executed really in parallel, not concurrently, copies of the function code are distributed to home memories of respective processor elements. Those copies of a function code are called clone functions. A clone function is a physical concept which is not included in the language syntax.

4.2. Multiplicity and sharedness of variables

In extending a serial language into a parallel one, new different concepts are added respectively to value passing and address passing. A variable passed by value to multiple function instances becomes independent function instance by function instance. Like automatic variables, they do not interfere with each other. On the other hand, a variable whose pointer is passed to multiple function instances is naturally shared by those function instances since the body which is pointed by respective pointers is only one. Thus the right to share the variable is virtually given when the address is passed.

4.3. Systematization of primitives

Starting from the function call, data passed by value and by address behave differently with the control of that function instance. Parallel primitives are com-posed into the above said two groups for the life of function instance. They are listed in Table 2. We refer to the language obtained by adding the syntax for primitives listed in Table 2 to lanaguage as C//.

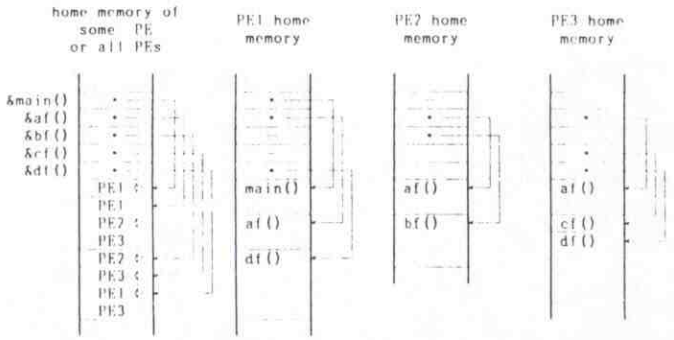

Fig. 2. Address acquisition table common to advance call and sequential call. Offset of the table entry serves as the pointer of corresponding function. Arrows in the leftmost table are the conceptual Illustration of the scheduling among clone functions.

Primitives in the value-passing group are related to the synchronization at the termination of control of a function instance as well as the return of a value from it. Those in the address-passing group are concerned with mutual exclusion or synchronization needed in variable sharing without extinction of a function instance.

The atomic lock-statement in the latter group covers a diversity of synchronization or exclusion logic. The among-statement in the former group supports nondeterministic or OR type synchronization in receiving a returned value from called function instances. A primitive for AND type synchronization is no needed because simple successive of reception of returned value by the // operator acts equivalently. Arbitrary combinational logic in reception or mixed synchronization logic of both groups is not supported since they are hardly needed and can be replaced by sophisticated programming, although **completeness** might be a little violated.

For functions, explicit template/instance expression is introduced into C//. For data, C// inherits the template/instance concept of the struct from C. Thus C// contains this concept for both control and data. However, a compound template of both data and procedure, that is, object-oriented abstract data type is not included in C//. It is not difficult essentially to introduce C++ type object expression as a successor of C//.

4.4. Expression of control parallelism

Methods of description of function call must be compatible with the language structures of the above

```
/* Son-Goku's Excursion Through a Labyrinth */
/* N, M and a maze pattern such as */
/*   int MAZE[N][M]={                                    */
/*       1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,···  */
/*       1,1,1,1,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,···  */
/*       0,0,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,0,1,1,···     */
/*       ···                                         };  */
/* are assumed to be defined externally.                 */

#define WALL 0
#define ROAD 1
#define GOAL 2
#define EXPLORED 3
#define DEAD_END 4
keyhole k[N][M];
int explore(int, int), enter_maze(void);

int enter_maze(void)
{   MAZE[0][0] = EXPLORED;
    return (explore(0,0) == GOAL);
}

int explore(int x, int y)
{   int i, j, xx, yy, ans, v[3], taskn[3], //task[3],
        vx[4]={1,-1,0,0}, vy[4]={0,0,1,-1};
    for(i = j = 0; i< 4; i++){
        xx = x+vx[i], yy = y+vy[i];
        if(xx>=0 && xx<N && yy>=0 && yy<M)
            if(MAZE[xx][yy] == GOAL){
                MAZE[x][y] = GOAL;
                return GOAL;
            } else lock(k[xx][yy]; k[xx][yy])
                if(MAZE[xx][yy] == ROAD){
                    MAZE[xx][yy] = EXPLORED;
                    v[j++] = i;
                }
    }
    if(j == 0) ans = DEAD_END;
    else if(j == 1) ans = explore(x+vx[v[0]], y+vy[v[0]]);
    else{
        for(i=0; i<j; i++)
            task[i] //= explore(x+vx[v[i]], y+vy[v[i]]);
        taskn[0] = 0, taskn[1] = 1, taskn[2] = 2;
        for(i = j-1; i >= 0 && ans != GOAL; i--){
            among(j; j <= i; task[taskn[j]]);
            ans = //task[taskn[j]];
                task[taskn[j]] = task[taskn[i]];
        }
        for(j = 0; j <= i; j++) /! task[taskn[j]];
    }
    if(ans == GOAL) MAZE[x][y] = GOAL;
    return ans;
}
```

Fig. 3. An example of recursive program (maze excursion).

(1), (2) and (3). In C// a function call is expressed in the following form called "*advance call.*"

```
t // = func(x);   /* calling        */
                  /*  a function    */

y = //t;          /* receiving      */
                  /*returned value  */
```

Mixed use of C compatible "sequential calls" with advance calls is allowed. The type of a call is distinguished only by the calling expression and there is no distinction in the called function. The environment of a sequentially called function is not a new heap but a conventional frame in the caller's stack in the old heap. A sequential call is good for compatibility and memory economy, but not for completeness.

The literal t in the above expressions is called a "*transfer variable.*" A transfer variable is a quasi-variable to which not a value but a conceptual control of a function instance is assigned by the operator "//=". Practically, it acts as the unique temporal name of the corresponding function instance. The operator "//" removes the control from the transfer variable and the corresponding function instance vanishes since direct assignment of a control without the //= operator or double assignment of controls to a transfer variable is not allowed and no function instance without its control held in any transfer variable can exist. The expression "*//transfer variable*" bears a numerical value and type equal to the returned value and type of the function instance. Evaluation of this expression is referred to as "reception" or "receive", and the execution of return statement of the side of called function instance is referred to as "*return.*"

The name of the transfer variable is declared with the type of a returned value from the function instance as "float //t" and so on. The scope of transfer variable is limited to being automatic.

When an operator // is encountered, return of the function instance corresponding to the operand is made to wait implicitly unless it has already been returned. Switch-spinning at this time is the job for the demand/accept control mechanism invisible form the program. Logically, a return value is always obtained by the // operator if and only if the operand transfer variable contains control of a function instance. On the other hand, the "/!" operator deprives the operand transfer variable of its control and the corresponding function instance is forcibly erased.

It becomes immediately clear that the advance call syntax satisfies (1) from examples such as

```
float x,y;
float f(float),z, //s,g(float),u, //t;
s //=f(x);
t //=g(y);
z=//s;
u=//t;
```

or

```
int k,x[M],f(int),y[M], //t[M];
for(k=0;k<M;k++)t[k]//=f(x[k]);
for(k=0;k<M;k++)y[k]=//t[k];
```

4.5. Nondeterministic selection of control

The primitive for nondeterministic selection of func-tion instances is the among-statement in which transfer variables are used to specify the corresponding function instances. Among-statement is written as

$$\text{among(j ; j<=1 ; j==0 ? s : t) ;}$$

or

$$\text{among(j ; j< M ; t[j]) ;}$$

where the first expression j in the parentheses is an integer or character type scanned one by one from zero until the value of the second expression becomes false or zero. The third expression specifies not a value but a control of the function instance for the respective value of j. Among those function instances that have been enumerated during the scan, one which has already returned is selected and the value of j is assigned. If none has returned, implicit synchronization occurs to wait for the earliest to return. Thus the earliest receivable function instance is selected through among-statement.

4.6. Mutual exclusion

The only primitive for mutual exclusion and synchronization independent of call, return or reception is lock-statement. In lock-statement as

$$\text{lock(g ; h)\{···\}}$$

g is locked just before the block in braces is entered and h is unlocked just after that block is exited. Variables g and h are "keyhole" type variables which follow the same scope rule as variables of other types. When a number of function instances access a shared variable x exclusively, mutual exclusion could be described with the aid of keyhole variable k in the same scope as x as

$$\text{lock(*pk ; *pk) *px=··· ;}$$

where pk and px carry &k and &k that might have been passed as arguments to function instances. Generally, the scope of a keyhole variable for this purpose should coincide with that of the shared variable. It might be very convenient to treat the keyhole variable in a stucture same as that of the shared variable.

Synchronous data passing from a callee to its caller is written using two keyhole variables g and h as

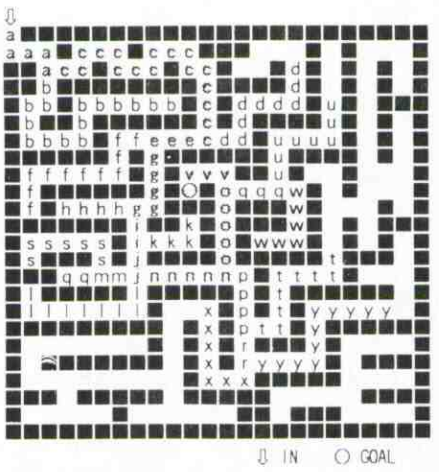

Fig. 4. Result of fig. 3 observed by adding printf() to the program.

```
/* caller side */
     lock (g;h) y =u;
/* callee side */
    lock (*pg; *ph) *pu = x
```

where pg, ph and pu contain the addresses of g, h and u respectively. It is assumed that g is locked and h is unlocked initially.

The well-known classical principle of composing general synchronization logic using private semaphore and global semaphore is also applicable in the lock-statement complex. In this cas private keyhole and global keyhole do not necessarily correspond to the linguistic scope.

A programming technique has been developed to encapsulate arbitrary synchronization logic ito a set of a struct containing all the keyhole variables required and functions which minipulate them through lock-statements. This set acts like a class of the object-oriented case, and an instance of which behaves like a classical monitor.

To restrict the time when x is exposed to a side effect of executing a function instance to receiving time only, delayed assignment operator "$" is made available. In an advance call, if an argument is passed in the form as

$$\text{t // =func(x \$) ;}$$

then x is passed by value once but change of x in func() is returned in synchronization with //t. The "variable folding" operation by delayed assignment could be regarded as the inverse operation of value passing by which a variable gains multiplicity.

4.7. Compatibility problems

In extending sequential language C to a parallel one, a few semantic mismatches reveal themselves in relation to the multiplicity of clone functions. The role of a function pointer, which does not lose its usefulness in C//, must be revised. It will be done in the next chapter.

An internal static variable has been thought in principle to belong to the function code, although some of today's compilers treat it differently. If there existed multiple internal static variables of the same name together with respective clone members which are invisible from the program consistency would not be maintained. An internal static variable must not be cloned in a way to an automatic variable which is shaped by all instances. It must be shared by all cloned members, and consequently by all function instances. Then the references to an internal static variable are mutually excluded using a keyhole variable in the internal static scope unless it is a read-only constant.

On the code level, the internal static area might be taken along with arbitrary one of cloned function members, and it might be accessed by absolute addressing mode, which is consistent with code copying.

Internal static variables are not so useful in parallel case as in C. The depth count of recursive calls does not correspond to the history of the function code. Data independent of the life of function instances should be secured not as code but as external struct since the latter can exist as an external structure.

5. Language to Machine Correspondence

C// programs do not describe the usage of processor elements. Programs are devised on the basis of logical parallelism. Since the function instances can be executed both concurrently or in parallel, a processor element may undertake several function instances at runtime if logical parallelism exceeds the number of physical processor elements. The cloned functions are distributed statically to multiple processor elements. Thus there arises multiple correspondences among function codes, processor elements, and function instances as shown schematically in Fig. 1.

A function instance can execute the code functions, of which one is selected properly at runtime. To contain the arbitrariness of the clone functions and to obtain defintie call address in execution time, a sophisticated method is introduced to the load module, which allows a solution of the problem of the function pointer mentioned in the previous chapter. A function pointer must be made to address any one of the cloned function codes. Meanwhile, in getting a call address, compatibility of an advance call and a sequential call must be maintained.

In addition to codes of functions, the linking loader generates tables like those shown in Fig. 2. All linked functions potentially subject to advance call are registered in these tables. The listings except the leftmost one correspond to the respective home memories of all the processor elements. For high-speed access, a duplicate of the leftmost table might be distributed to processor elements except for scheduling pointers. Offsets of entries for the same function are made to be same among all listings shown, letting a function pointer be that offset address. The code performs memory-indirect addressing of a sequential call, referring to the table for its own processor element. In an advance call, the code refers to the leftmost table to determine which processor element to demand before gaining a function code address in that processor element from the corresponding table.

In the present demand/accept type computer, function codes are loaded statically. Then, which function codes including cloned function codes are allocated to which home memories must be determined. At the present stage the code allocation is specified by a person in "*load conditioning file*" outside the logical C// program, which is fed to the linking loader. The default specification is to make as many clones as processor elements for the respective functions and load clones of all functions to all processor elements uniformly.

On the other hand, distribution of function instances to cloned functions is done at runtime. The scheduling strategy then might be round-robin, random, processor load balancing, access cost balancing with respect to external data and so on. Nevertheless, the optimal strategy is beyond our knowledge and depends on both machine parameters and program profiles. Migration during the life of a function instance is not assumed here since the transportation of its environ-

ment would be costly compared to load unbalance of processor elements when the number of processor element becomes large.

The optimization problem discussed in 3. (III) remains outside of C// but is closely related to C//. It includes the automatic generation of the optimal load conditioning file and optimal spatial-temporal scheduling of function instances.

Initially the C// processor is implemented as a preprocessor to C, but is later replaced by an experimental cross-compiler since the former cannot cover all syntax and generates a very heavy code for an advance call, decreasing the efficiency of the demand/ accept control mechanism. The target machine of the compiler is a parallel computer based on the Motorola 68000 family processor chips with the demand/accept control mechanism emulated through the emulation trap.

The conventional technology using yacc and lex has been sufficient for compiler realization. No syntactic or lexical problems have arisen. As discussed before, the environment of a function instance is a heap within which stacks are stacked. As the transfer variables are declared before advance calls, frames for parameter passing can be prepared on the stack. Heap to heap parameter passing can be realized by either data copyingt or frame pointing passing. Our experimental compiler adopts the latter.

6. Examples and Experience

According to the objective of this language, its evaluation should be done both in adaptability ot machines and in appropriateness for the applications assumed in 2. The former is discussed in the foregoing chapter. For the latter, there is noi appropriate method to achieve the above but, for empirical verification, it is accomplished by many programming examples. Though the empirical research does not in general lead to general and quantitative conclusions, it may give insight not only into the appropriateness of the language for applications but also into its affinity to human programmer's ideas.

At first, a few tens of maze examples were made to develop programming techniques and styles. One of those programs is shown in Fig. 3 which is titled after Son-Goku, a monkey hero in an old Chinese fairy tale who can produce his copy from his hair. However, the tale does not tell us whether he can do this recursively or the context of the program does so by an advance call of explore(). Function instances of explore() return a report whether the branch leads to the goal, which is received in the order of arrival through the among-statement. The monitored result of the program plotted on the maze pattern is shown in Fig. 4.

Qualitatively, the among-statement was used more often than expected in small examples. Some programming techniques were developed on the basis of that statement including "repacking" and "dependence-graph tracing." Using the among-statement also, diffusive recursion can be converted into centralized control just as recursion can be rewritten in a loop in a sequential case. Nondeterministic selection has extended the range of programming paradigm considerably. This result should be fed back to the performance improvement of the compiled code and control mechanism. On the other hand, the lock-statement was used on the whole in a very simple context as expected. More highly equipped synchronization primitive than the atomic one was not required. Solutions to classical exclusion and synchronization problems such as the procducer-consumer, five philosophers and the like were described using the lock-statement, but eractical applications do not exist.

Evaluation by small toy programs is far from being practical especially from the point of view of the factor of human programmers. We selected a graduate student who had a full knowledge in picture processing and programming in language C but had not touched the design and implementation of C//, and made him write an actual picture analysis program in C// according to the algorithm reported in [15] from the gradient computation to the regulation of "space list."

All stages of the program belonged to type (A), in which only the gradient computation by convoluting differentiating filter was SIMD compatible through which strip areas of the picture were passed to function instances. At an other stage, the processing load was divided into function instances for edge segments or those for space-list segments. Length or structure those segments were nonuniform. Respective stages ran 3.1 to 3.9 times faster on the parallel computer with four processor elements. Type (C) strategy was also tried and two or three independent control flows were possible in one half of processing stages to which (A) was not used. Lock-statements were used in simple ways in getting new list cells exclusively.

7. Conclusions

The design, specification, implementation and pro-

gramming experiences of a parallel programming language C// are presented. The language is designed to conform to the shared memory parallel computers characterized by the distributed "demand/accept" control mechanism built into all processor elements. The first objective of C// is to make those parallel computers widely applicable. An optimal strategy for spatial-temporal scheduling of function instances is an open problem. Affinity to human programmer, is also considered in design and evaluated empirically by programming experiments.

It is apparent from the language specifications that, C// could be compiled in principle for other types of parallel computers. However, if a target computer operates based on a centralized control system or on heavy communication channels, effective execution of C// programs cannot be expected. In this sense, C// is machine-oriented. But this does not mean that programs written in C// automatically become optimal with to respect parallel computers with the demand/ accept control. A programming language is only a mediator. If a language yields any new styles of programming, they should be fed back for the machine design.

On the other hand, parallel languages of various types are possible for computers with the demand/ accept control other than the call type C//. For example, a programming language based on message passing between static processes could be realized by locked communications through shared data between processes created initially and looping infinitely without DELIVERing. A programming language that adopts a directed graph as its control flow model could be realized by the runtime support of resident shells distributed to processor elements that manage what to fire by DEMAND next. Among those language models, object-oriented one is very promising not only in terms of programming paradigm but also space scheduling since it binds everlasting data instances with procedures which access them. The object-oriented concept will be introduced to a second programming language for parallel computers of the same type in the near future.

Anyhow, the most urgent work to be done for parallel computers is gaining practical programming experience.

REFERENCES

1. S. Oyanagi and N. Tanabe. Realization Technologies for Massively Parallel Machines, J.IPS Japan, Vol. **32**, No. 4, pp. 365-376 (1991).
2. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing System, ACM Comput. Surv., Vol. **21**, No. 3, pp. 261-322 (1989).
3. Y. Hirotani, A. Fukuda, K. Murakami and S. Tomita. The Parallel Programming Language SERVE, I.E.I.C.E. Technical Report Japan, Vol. **89**, No. 410, pp. 1-9 (1990).
4. M. Hirabaru, K. Araki and I. Arita. Design and Implementation of the High-Level Parallel Programming Language Nano-2, Trans. on I.E.I.C.E. Japan, Part D, Vol. **J71-D**, No. 8, pp. 1518-1524 (1988).
5. G. L. Steele Jr. and D. W. Hillis. Connection Machine Lisp: Fine-Grained Parallel Symbolic-Processing, Prc. of the 1986 ACM Conference on Lisp and Functional Programming (1986).
6. A. Goto. Parallel Inference Machine Architectures, J.IPS Japan, Vol. **32**, No. 4, pp. 458-467 (1991).
7. T. Yoshinaga, M. Suzuki, T. Teraoka, H. Mogi and T. Baba. A Node Processor for the A-NET Multiprocessor and Its Execution Scheme, Joint Symposium on Parallel Processing '91, pp. 189-196 (1991).
8. A. Agarwal, B. H. Lim, D. Kranz and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing, Proc. 14th Ann. Symp. on Computer Architecture, pp. 104-114 (1990).
9. R. Onai. Occam and Transputer, Kyouritsu Pub. (1986).
10. M. Tomisawa. Demand/Accept Control Mechanism and Hardware of a Parallel Computer, Joint Symposium on Parallel Processing '91, pp. 221-228 (1991).
11. E. C. Cooper. C Threads, Computer Science Dept., Carnegie Mellon Univ., CMU-CS-88-154 (1988).
12. M. tomisawa, S. Igarashi, O. Atoda and N. Saito. The Distributed Built-in Control Mechanism for Procedure Flow Multiprocessors, Trans. I.E.I.C.E. Japan, Prat D, Vol. **J71-D**, No. 8, pp. 1921-11930 (1980).
13. T. Nakagawa and M. Sugie. Automatic Load Ballancing Features on PIM/c, I.E.I.C.E. Technical Reports Japan, Vol. **91**, No. 130, pp. 41-47 (1991).
14. T. Suzuki, M. Tomisawa, S. Igarashi and O. Atoda. The Past Sharing/Multiple Future Data and Caching Mechanism., IPS Japan SIG Notes, Vol. 91, No. 64, pp. 177-184 (1991).
15. A. Ogasawara, M. Oide, O. Atoda, S. Igarashi and N. Saito. A Functional Analog of Motion Parallax in Machine Vision, Trans. I.E.I.C.E. Japan, Part D-II, Vol. **J74-D-II**, No. 7, pp. 933-944 (1991).

AUTHORS (from left to right)

Masaki Tomisawa. Education: Tokyo University of Agriculture and Technology, B.S., 1987, M.S., 1989, D.Eng. 1992. Academic Appointments: Tokyo University of Agriculture and Technology, Assistant.

Hitoshi Tamura. Education: Tokyo University of Agriculture and Technology, B.S., 1990, first phase of doctoral program completed, 1992. Industry positions: CASIO Computer Company, Ltd., 1992.

Hitoshi Hoshino. Education: Tokyo University of Agriculture and Technology, B.S., 1990, first phase of doctoral program completed, 1992. Industry positions: Yokogawa Electric Corporation, 1992.

Satoshi Igarashi. Education: Tokyo University of Agriculture and Technology, B.S., 1981, M.S., 1983; D.Eng. Academic appointments. Tokyo University of Agriculture and Technology, Assistant, 1983, Associate Professor, 1991. Memberships: Information Processing Society, Japan Society of Instrument and Control Engineers.

Oichi Atoda. Education: Tokyo University, B.S., 1970, M.S., 1972, D.Eng., 1975. Academic appointments: Tokyo University, Assistant, 1975, Lecturer; Tokyo University of Agriculture and Technology, Associate Professor, 1978, Professor.

Nobuo Saito. Education: Tokyo University, B.S., 1955; D.Eng. Academic appointments: Tokyo University of Agriculture and Technology, Professor, 1978. Industry positions: Hitachi, Lt. (Central Research Laboratories), 1955. Research interests: ferrites, magnetic thin film, plated wire memory, magnetic bubble memory.