Regular Paper

A Local Operating System for the A-NET Parallel Object-Oriented Computer

TSUTOMU YOSHINAGA* and TAKANOBU BABA*

The A-NET local operating system is designed to support a parallel object-oriented execution model. It is placed on each node of a multicomputer and provides several mechanisms such as message handling, context scheduling, and object management. Its major design principles are a quick start of the method execution and low-overhead context switching. To attain these goals, we use message selectors which include absolute addresses to reach the methods, and two sets of special registers. Our experimental results show that (1) the ratio of executed machine cycles between user and system is approximately 1:10, and (2) we can reduce the method searching cost by on an average of 2.9 times, by using addressed selectors.

1. Introduction

To be able to cope with the need for increased computation power, a lot of research has been done to design and develop parallel computers. These parallel computers tend to include more than a thousand nodes with improved node potential and memory capacity [1]. We need *parallel operating systems* to utilize effectively the performance of these highly parallel computers.

The parallel operating systems for distributed-memory, MIMD, parallel computers are divided into the following two groups: (a) general hardware independent, and (b) specialized machines that are well suited to their execution model. Table 1 is a summary of the parallel operating systems, machines and their features. For (a), several machines use UNIX based system like NCUBE2 [11]. These are similar to a distributed operating system for workstations, connected to a local area network. NX/2 [8] for the iPSC/2 and Reactive Kernel for the Ametec Series 2010[10] also belong to (a). These systems are suitable for controling relatively coarse-grain parallel processing. PIMOS [9] for the PIM, and COSMOS [6] for the J-Machine belong to (b). They are designed to execute a parallel logic programming language KL1 and a concurrent object-oriented language CST, respectively. Therefore, these design approaches are execution model and language combined.

The A-NET local operating system (hereafter referred to as local OS) belongs to (b), and its goal is to support efficient execution of a parallel object-oriented model [2, 14]. Execution performance is very important for a parallel operating system so that we have to make clear some technical problems, such as low-overhead message driven mechanism, synchronization between objects, and difficulties in supporting dynamic program structure with creation and deletion of objects. The model is similar to the COSMOS's model. However, the local OS is applied from medium to coarse-grain parallelism that is brought by the user defined objects. While the COSMOS's model which is suited to data object level fine-grain concurrence. The local OS is placed on each processing element (PE) and provides several functions, such as message handling, context scheduling, and object management. We say "local" because, all the nodes include the same operating system code, and its services are limited to inside the allocated PE.

In this paper, we describe an overview of the A-NET project and fundamental functions of the local OS. Then, we show some experimental results.

2. An Overview of the A-NET

2.1 A Parallel Object-Oriented Language A-NETL

We designed A-NETL [15] keeping in consideration of a trade-off between its functions and the hardware cost of its implementation. It includes quite a few message expressions and constructs to extract parallelism, whereas it has a few constructs of sequential controls and data structure. We summarize its features as follows.

(1) Description for static load distribution

The A-NETL has three explicit declarations, *group*, *collect*, and *link*, to express inter-object logical configuration that will be reflected in physical mapping.

*Department of Information Science, Utsunomiya University, Utsunomiya, 321, Japan.

Table 1 A summary of the parallel operating systems.

Parallel OS	Machine	Node	The Number of Nodes	Memory /Node	Features
UNIX SV	NCUBE2	64 bits CPU, FPU	8,192	1-64 MB	Each computing element executes independent UNIX kernel.
NX/2	iPSC/860	i80860	127	8-16 MB	It supports a more streamlined and flexible set of message passing service calls.
Reactive Kernel	Ametek 2010	M68020 M68882	512	−8 MB	A small kernel that dispatches to handlers according to the tag in the message.
PIMOS	PIM	40 bits original	512	−80 MB	It is designed to run a concurrent logic programming language KL1.
COSMOS	J-Machine	36 bits MDP	65,536	4 KW+1 MW (36 bits)	It is designed to efficiently support fine-grain actors.
A-NET Local OS	A-NET Machine	40 bits original	order of thousand	−320 KB	It is applied to medium to coarse-grain parallelism.

The *group* declaration is convenient for bundling a group of related objects. The other two are used to indicate physical allocation. The *collect* declares that collected objects should be allocated to the same node. The *link* declares the strength of the relationship between objects or groups so that they can be allocated to close nodes. We have also developed an *allocator* which supports the fashion of *network-topology independent* object allocation[3]. To use the function of static load distribution efficiently, A-NETL has a definition of so-called *indexed objects* to create multiple objects statically. The A-NETL compiler creates the indicated number of objects with the same methods.

(2) Message passing expression

The A-NETL has three types of message sending methods, i.e., past, now, and future, as does ABCL [13]. For each type, a user can use a multicast mode that sends the same message to multiple objects. The message expression is executed by the PE's high level machine instruction. One message expression corresponds to one machine's instruction basically. There are 9 message-sending instructions out of 68 user's machine instructions. They include return, dynamic object creation, and dynamic method appending and changing. There is another function, *Multi-receive*. Here the receiver may wait for multiple messages before starting a method execution. It is convenient to describe a method that needs multiple inputs. In this case, arguments may be constructed as an ordered list. The A-NETL allows a user to control the execution sequence by using these synchronous and asynchronous message passing constructs.

Figure 1 shows a quick sort program described in A-NETL. This program consists of 127 indexed objects, *quick* [*i*], ($i=1$ to 127). A *link* declaration means that these indexed objects make a task graph of a binary tree. When a message with a list of some data *sort: data* is sent to the *quick [1]*, it checks the size of the list. Then if there are fewer than four elements in the list, the object sorts the data by itself. If there are four elements or more in the list, it finds its subordinate objects and passes half of the data to each. When subordinate objects don't exist, it creates two dynamic objects. Each subordinate object repeats the same process.

2.2 Hardware Organization

The A-NET computer is a distributed memory, MIMD, highly parallel computer. Its node processor [16] consists of a PE for executing methods, a router for message passing, and a memory of 320 KB, as shown in Fig. 2. Its routing algorithm is changeable so that several kinds of interconnection networks may be realized.

The PE for a prototype machine is a microprogrammed processor, controlled by horizontal microinstruction of 73 bits. The major units include 32 bits of an arithmetic-logic unit (ALU), a floating point processing unit (FPU), 8 bits of a tag processing unit (TPU), an instruction preprocessing unit (IPU), 2 sets of special registers (SR), and maximum 52 K words 40 bits per word of local memory (LM). The TPU is used to check a flag, which represents whether a return value for a future type message is stored or not, and to compare the data types. The IPU supports the variable length code fetch, and the base addressed operand processing. Two sets of the SR, for system and user, reduce the overhead of context switching which is caused by message arrival.

The router provides packet switching using adaptive virtual cut-through routing and compiled object code transfer using circuit switching. The reasons why we selected adaptive virtual cut-through routing are (i) the organization of the router should be independent of the network-topology, (ii) most of the messages are too small to send by circuit switching, and (iii) adaptable routing is more flexible to avoid deadlock than is nonadaptable routing. The major reason why we use circuit switching together is that an object code is much bigger than a message and transferring an object code

```
#define     MAX           127                                           " number of the indexed objects "
#define     N             MAX / 2
link        quick[1--N]   ( quick[j = 0--1:  2*selfIndex + j] )         " relation for the binary tree "
extern      classOfQuick.                                               " external reference "

object      quick[MAX]                                                  " definition of the object name "
methods (                                                               " method starts from here "
      sort: data (                                                      " data are list of the data to be sorted "
         ¦ sortedData noOfData min left right leftData rightData k p ¦  " temporary variables "
         sortedData with: nil.                                          " initialize a list to store sorted data "
         noOfData = data size.                                          " number of the data to be stored "
         if (noOfData < 4) then (                                       " each datum is a leaf "
               while ( (data size) <> 0) (                              " sort less than 4 data "
                     min = data first.                                  " initialize minimum "
                     data do: [:each ¦ if (each < min) then min = each. ].
                     sortedData addLast: min.                           " search the minimum from the data "
                     data remove: min.                                  " remove the minimun from the data "
               )     " while "
               ! sortedData.                                            " return the sorted data "
         )     " if "
         else (                                                         " each datum is not a leaf "
               if ( selfIndex  < N) then (                              " check whether this object is a leaf or not "
                     k = 2 * selfIndex.                                 " there are subtree under this object "
                     left  = quick[k].                                  " decide the two indexed objects "
                     right = quick[k+1].                                " which are allocated "
               )                                                        " on the children nodes "
               else (                                                   " self is a leaf,     "
                     left  = classOfQuick new.                          " so create two dynamic objects "
                     right = classOfQuick new.                          " to share the subtasks "
               )
               leftData   with: nil.                                    " initialize the list of data to share "
               rightData with: nil.
               p = (data first) + (data last) / 2.                      " caluculate the middle value "
               data do: [:each ¦ if (each <= p) then                    " share the data into two lists "
                          leftData   addLast: each.                     " values less than p are to the left "
                     else rightData addLast: each.                      " greater values are to the right "
               ].
               leftData   = left sort: leftData.                        " ask to sort the half of data "
               rightData = right sort: rightData.                       " into two children objects "
               while (leftData <> nil)                                  " concatinate the lists "
                     rightData addFirst: (leftData removeLast]
               ! rightData.                                             " return the combined list "
         )        " else "
      )        " sort: "
)          " end of the methods "
```

Fig. 1 An A-NETL sample program for quick sort.

occurs less frequently than message passing. The initiative of data transferring is taken by the sending port, using an FIFO in the receiving port as a buffer storage.

The PE and the router share maximum 12 K words, 40 bits of common memory (CM). In this memory, input/output message queue, shared values between the PE and the router, and the storage for packet assembling are maintained.

3. A-NET Local Operating System

3.1 Design Principles

We designed the A-NET local operating system (local OS) to achieve efficient parallel object-oriented execution model of the A-NETL such as synchronous and asynchronous message passing. The design principles of the local OS are as follows:

(1) Fast message dispatching

In the A-NET, a unit of parallel processing is the

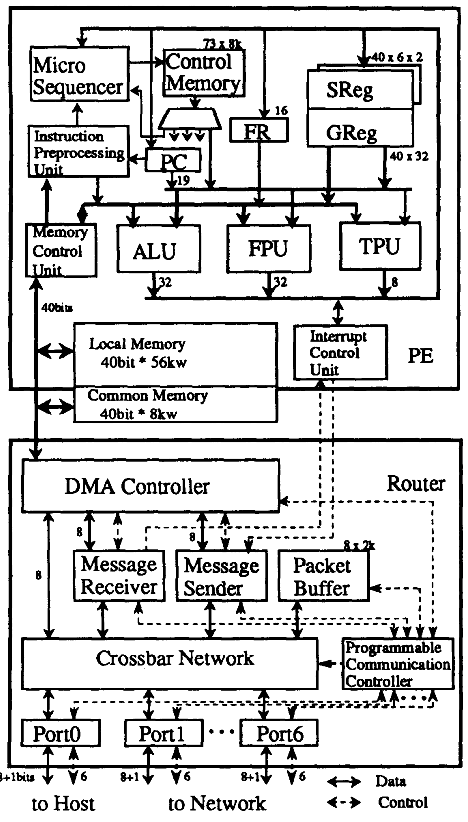


Fig. 2 Hardware organization of a node processor.

method execution, activated by a message. Thus, it needs to start the execution of a user's method as quickly as possible after message reception.

(2) Low-overhead context switching

A grain size of massively parallel processing tends to be small. To reduce the overhead of subdivision of the execution, the context should be switched quickly.

(3) Local management

An object may be created, exchanged, and killed dynamically. Therefore, it is difficult for all the nodes to maintain global information. On the other hand, centralized control is not good either, because it brings a bottleneck. So at first, we don't introduce dynamic object migration except for the explicit one based on a user's description. The present local OS manages only the objects allocated on each PE.

(4) Debugging support

It provides functions to debug parallel programs such as inspecting the states of an object, and saving sent/received message histories to show a timing chart.

3.2 Basic Functions of the Local OS

In this section, we describe some ideas for fast and low-overhead operation, and its implementation based on the above design principles.

3.2.1 Message Reception Mechanism

When the router receives a message on a destination node, it stores the message into the input message queue on the CM, then it interrupts the PE. The local OS, invoked by the message interruption, reads the messages one by one.

The ideas for fast receiver message dispatching are as follows:

• The object ID, which becomes an address of a message, contains an index of an object table for the receiver as well as a node number. Therefore, it is easy to check whether an object is alive or deleted.

• The message selector for a static object includes an absolute address to find out method pointer. The local OS needs to search a message dictionary only for the message selectors of dynamic objects.

• It copies the arguments of a received message from the CM to the LM like one array without unfolding.

• A method execution for a received message is started without creating a context.

The local OS is described in the A-NETL as a single object, called a system object. It is a collection of the methods for system services. Figure 3 shows an abstraction of the system data structure and a message reception.

In Fig. 3, a system method for receiving messages is started by using an *interrupt vector table.* The contents of the interrupt vector table are the *addressed selectors* for the system methods. The addressed selector points the *message information* that includes a method pointer, information of the arguments, and the information whether it needs to wait for the other messages before starting its execution or not. Firstly, a user object corresponding to the message receiver is confirmed in its existence by checking an *object reference table.* Then a method that corresponds to the message selector is searched. If there are message's arguments, they are copied onto the LM and removed from the queue. When there is structured data in message's arguments, the local OS needs to relocate the pointers with copying. Each structured data has one word of the data entry that include data type and size. The arguments are packed into a message so that all data entries are listed first, then data cells follow. This structure isn't needed to copy the cells one by one with understanding the data types. After that, the local OS executes the searched method or the method executed before the interruption. The Method execution from the received message is performed by a privileged *receive* instruction. It started after setting the user's special registers (UR) and a program counter (PC) without creating a context. When an interruption occurs while carrying out the user's execution, its image isn't stored so that it can only be reactivated by setting a flag register (FR) and the PC.

3.2.2 Context Management

The context sheduling needs to reflect the parallel object-oriented execution model. Our ideas for low-

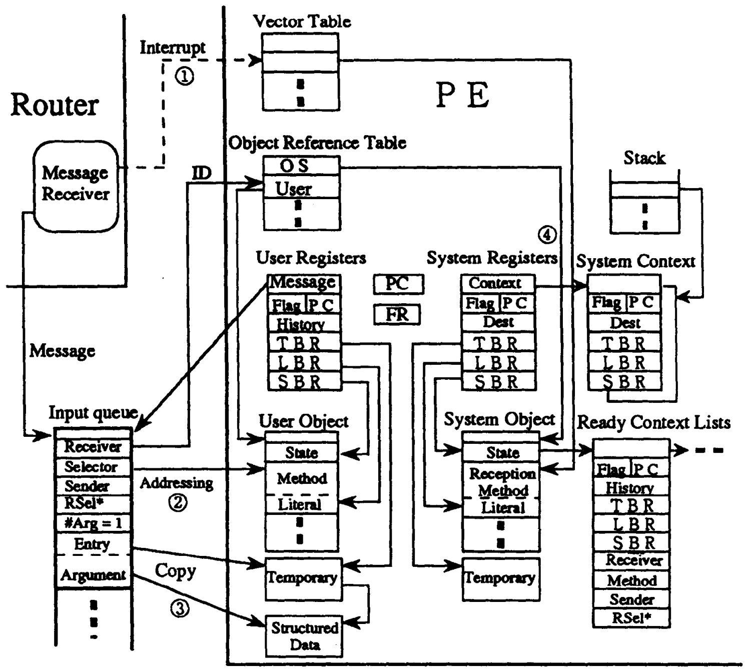


Fig. 3 System data structure and a message reception mechanism.
*Rsel: Return Selector

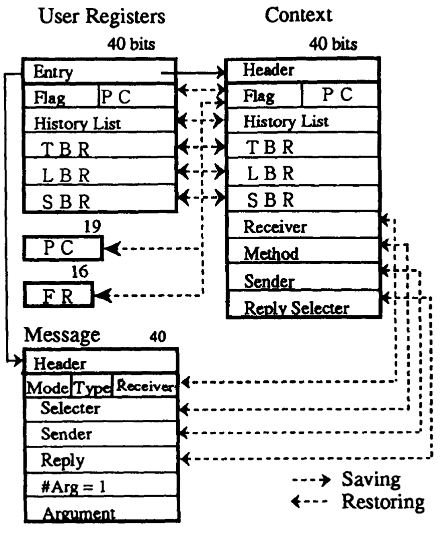
overhead context switching are as follows:

• Using two sets of a few special registers and memory oriented machine instruction set, we reduce not only the number of times to store an execution image but also the amount of the image that should be stored and restored.

• A synchronization for the future type message is performed by an exception process. It is hardware supported and caused when the TPU checks a flag for each data.

When a received message needs to wait for another message's arrival before starting method execution, a context is created to control its waiting. When a received message is a reply, the local OS activates contexts, which have been suspended for synchronization, using a *return dictionary (RDict)*. The *RDict* is made when a now or a future type message is sent.

A variable that is to be stored a return value in it is called a *future variable*. When the *future variable* is accessed before the arrival of the return value, context switching occurs. It is called a *future trap*. In order to use a block transfer for saving and restoring an execution image, the structures of the UR, the context and the message are unified. Figure 4 shows these structures and their relation to saving and the restoring context. A context, which is stored by a future trap, is listed in a return dictionary and suspended until an arrival of a required reply.



TBR : Temporary Base Register
LBR : Literal Base Register
SBR : State Base Register

Fig. 4 Saving and restoring context.

3.2.3 Memory Management

We selected generation scavenging [12] for storage reclamation. Figure 5 shows a memory map of the LM and the CM. A heap area, which can be used in execution time, is divided into three parts for the "garbage" collection.

User defined objects are allocated in an *old area* from the beginning, because they were rarely deleted from our initial programming experience. An allocation and deletion of a user defined object is performed to book and erase to/from the *object reference table*. To reduce the overhead for relocation, we use self-relative addressing for branch addresses and the base addressing for operand addresses. Therefore, the local OS needs to relocate only the pointers included in a literal frame because we use 16 bits of absolute address for the speed-up of struct operation.

Data objects that are created in a *new area* during the execution, the things such as cells from a list or an array, are scavenged to a *future area*. Then older data objects are moved to the *old area* to reduce the time spent by scavenging more stable objects. This function is supported by the firmware to reduce pause time.

The CM is used with input and output message queues, and common information between the PE and the router such as queue pointers, the number of allocated user objects, and the size of available areas. The major reason why we selected this memory structure is that the router can access messages or usage information of the LM without stopping the action of the PE.

3.2.4 Debugging Support

To make good use of multiple nodes and to prevent making a "hot" spot, we selected a distributed debugging model.

The A-NETL debugger consists of a *master*, which works on the host for user interface, and *locals*, which work on each PE as traditional parallel debugger [4]. The traditional parallel debugger means collection of sequential debuggers that use breakpoints and tracing [7]. We also use an event history that corresponds to message sending, receiving, and context switching. The *locals* consist of two levels, local OS and firmware. The local OS level ones provide recording of received messages with time stamps, setting conditions for breakpoints, communication with the *master*, and switching between normal execution mode and debug mode. The *locals* use local, logical time stamps because the A-NET machine doesn't have a global clock. The *master* displays global relations between objects after retrieving local event histories, whereas a user can set onto the debug mode per PE.

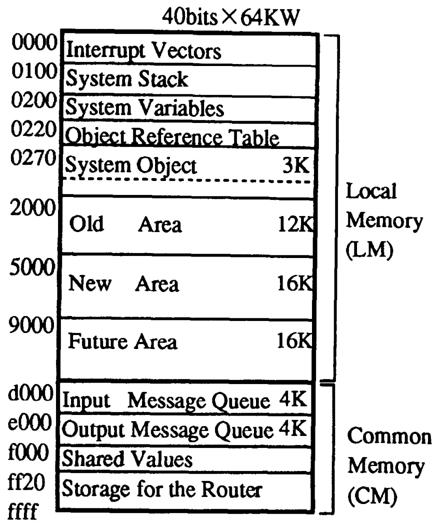


Fig. 5 Memory map.

4. Experimental Results

Now the local OS version 2 is available on a software simulator [5]. To verify total balance between system and user software, and hardware support, we have made an experiment using the simulator. The local OS version 2, which is used for the experiment, consists of 63 methods, 15.5 KB, and it occupies approximately 25.5% of the *old memory area.*

Table 2 shows the results when we used a boolean n-cube. we discuss about an execution grain size per message and the effect of the ideas for fast message dispatching in (1). Then we evaluate the cost concerned to context switching in (2). (3) describes a parallelism and an execution balance between system and user. Finally, we evaluate the memory capacity in (4).

(1) Processing grain size and speed for a message

An average number of user's machine cycles which are executed per message interruption is 398.7 (1 machine cycle = 125 *ns*). It is a fairly small task that corresponds to 20 to 30 machine instructions. Figure 6 shows a process that is executed after a message interruption. In the figure, the arrows show the call and the return of the system method, and the numbers express the executed machine instructions and cycles in a typical case. A user's method is executed at the broken line.

The time that is needed to receive a message depends on the number of arguments and their data types. The average number of arguments per message is 1.4 from Table 2. In the case of a message which has one argument, the speed up from version 1 was 17 to 49% [14]. This improvement is brought by object/method searching using the object reference table and addressed selectors, message dispatching without creating a context, and copying of message's arguments described in Section 3.2.1. However the local OS spends more than

Table 2 Simulation results.

Program	TSP with neural network	Color matching puzzle	Simulation 1 of chemical reaction	Simulation 2 of chemical reaction	Binary tree database	Logical Simulation of a sequencer	Matrix calculation	Average
The number of total object	32	194	6	14	15	28	21	44.3
Statically/Dynammically	7/25	2/192	6/0	3/11	15/0	5/23	5/16	6.1/38.1
Average object size (wds)	297.7	343.2	187.3	210	209.1	67.9	163.8	211.3
Average number of methods	7.8	6.7	4	5.4	8.3	2.0	6.7	5.8
The Number of used PEs	16	16	6	14	15	16	16	14.1
Execution machine cycles								
User	3,621,249	2,728,298	129,351	342,880	116,728	61,003	72,715	1,010,318
System	23,535,364	26,868,085	1,568,039	12,170,185	3,045,353	2,765,152	1,871,672	10,260,550
User:System	1:6.5	1:9.8	1:12.1	1:35.5	1:26.0	1:45.3	1:25.7	1:10
GC/Node	1.7	2.6	0	1.4	0.1	0	0	1.4
Message Interrupt	3,634	4,305	289	2,024	558	582	255	1,663.9
Size (wds)	8.2	6.1	6.7	5.9	5.6	5.68	6.45	6.38
The Number of arguments	2.3	1.5	1.6	0.7	1.0	0.9	1.7	1.4
User/Message (Machine cycles)	996.5	683.2	342.9	169.4	209.2	104.8	285.2	398.7
Future trap	111	2,365	0	22	168	25	44	390.7
Maximum parallelism	15	14	5	11	10	8	9	10.3
Average number of active nodes	6.8	8.0	0.9	5.2	1.9	4.1	2.1	4.14

PE: 1 machin cycle=125 ns, Local memory: 1 word=40 bits.

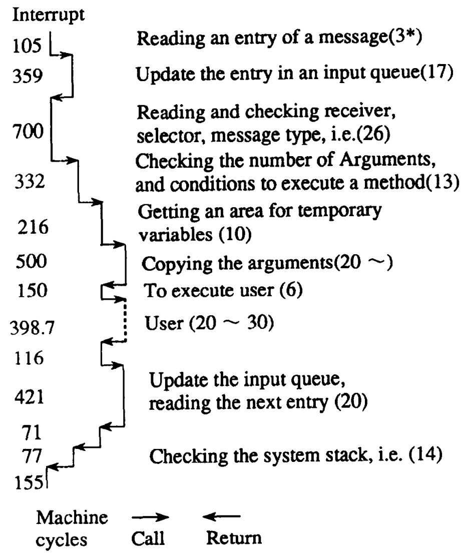

Interrupt
105
359
700
332
216
500
150
398.7
116
421
71
77
155
Reading an entry of a message(3*)
Update the entry in an input queue(17)
Reading and checking receiver, selector, message type, i.e.(26)
Checking the number of Arguments, and conditions to execute a method(13)
Getting an area for temporary variables (10)
Copying the arguments(20 ~)
To execute user (6)
User (20 ~ 30)
Update the input queue, reading the next entry (20)
Checking the system stack, i.e. (14)
Machine cycles
Call
Return

Fig. 6 A message reception detail.
*the Number of executed machine instructions

4000 machine cycles per message reception. This means that the local OS spends about ten times longer than fine-grain user's execution. The ratio of total executed machine cycles between the user and the system is 1:10 in the average. It is equal to the ratio of the grain size between the user and the system per message.

The average number of methods included in an object is 5.8. It means if a user defines the indexed object, addressed selectors reduce 2.9 (=5.8/2) times message dictionary searching in average.

(2) Cost of the future trap and context switching

The local OS executes approximately 700 machine cycles for a future trap. The smaller the grain size of a user's execution becomes, the more likely future traps happen because the interval time between message sending and reference of its return value becomes short [2].

The context switching between user and system also occurs when the message interruption takes place during user's program execution. The local OS makes it low-overhead using two sets of special registers. It is performed 16 machine cycles (only to store user's PC and flags) plus 50 machine cycles (starting system's execution) in order to switch user to system, and 14 machine cycles to restore user's execution image.

(3) A relation between system and user program

In the Table 2, maximum parallelism means the maximum number of nodes that are executed simultaneously except for initialization. As the simulator doesn't allocate dynamic objects on its class node, almost all the number of used PEs are executed at the same time at the peak time.

Figure 7 shows the processes when we execute a traveling salesman problem with the boolean 5-cubes. The vertical axis expresses the number of executed PEs and the horizontal axis expresses a real time of the A-NET machine. Three graphs show (a) total, (b) system, (c) user respectively. In this example, 25 objects for neurons are defined using the indexed objects. After the allocation of the user objects and their initialization, an evaluation function is calculated six times. Then the evaluated value is reduced. We notice that the activity of the system varies according to the user's algorithm. They reach the peak throughout its execution time while the average numbers of the executed nodes are (a) 9.9,

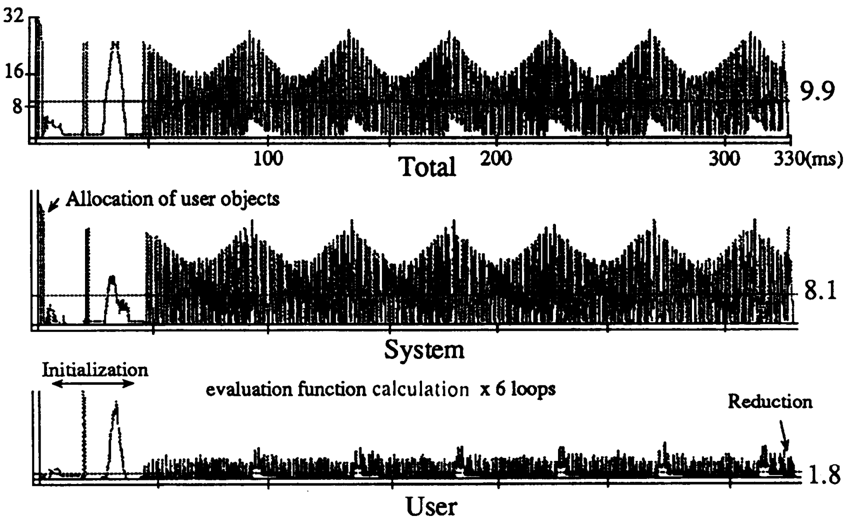


Fig. 7 Execution process of a traveling salesman problem.
This program is solved by a neural network using the Hopfield model. It includes 25 objects which correspond to the neurons to make completely connected network for 5 cities. We used a boolean 5-cubes network topology which includes 32 nodes for above simulation.

(b) 8.1, (c) 1.8, respectively.

(4) Local memory capacity

The consumption of storage space depends on application programs and their algorithms. The "garbage" collection happened 0 to 4 times on each node when we executed such sizes of programs. Therefore, there aren't so many persistent objects that survive for several generations. We can conclude that the size of the new and future area is enough. We can also estimate that 211.3 words of 58 average user defined objects are able to be allocated in the old area. The number is enough to use *collect* declaration which is described in Section 2.1.

The average size of a message is 6.38 words, and the average number of its arrival is 1,663.9 times for all the nodes. It means that they spent 2.4 cycles of 4 K words of cyclic input message queue.

5. Conclusion

We described an overview of the A-NET project, the design of the local OS, and the experimental results.

From the evaluation, we found that the system spends its execution time about ten times longer than the average user. In the case of COSMOS, an average fine-grain program spends about 70% of its active time in the operating system and 30% of the time in user code by using the hardware task scheduling and dispatching mechanisms of the MDP [6]. One reason of the system overhead is brought by a high-level user interface of the object-oriented model.

We are considering two minor changes of execution mechanisms to improve the ratio. One is the improvement for message reception mechanism. If the router puts an arriving message into the LM directly, the local OS doesn't have to copy its arguments from the CM to the LM. To adopt this mechanism, we have to review the load balance and role sharing between the PE, the router, and the local OS. Because the router has to cease the activity of the PE when it writes a message, we have to set up so that the router writes a message just after the PE finishes the previous method execution for typical programs. Another idea is to use calling sequence without executing the local OS when a user's program sends a message to call a method on the same node.

We are planning the next version of the local OS that will support a file system, and a master debugger using a graphical user interface. The evaluation of the debugging function is also future work. After that, we intend to make an experiment with a highly parallel multicomputer using a network-wide simulator.

References

1. Athas, W. C. and Seitz, C. L. Multicomputers: Message Passing Concurrent Computers, *IEEE Computer* (1988), 9-24.
2. Baba, T. et al. A Parallel Object-Oriented Total Architecture: A-NET, *Proc. Supercomputing '90* (1990), 278-285.
3. Baba, T. et al. A Network-Topology Independent Task Allocation Strategy for Parallel Computers, *Proc. Supercomputing '90* (1990), 878-887.
4. Hamada, M. et al. Programming Debugging Scheme for a parallel Object-Oriented Total Architecture A-NET (in Japanese), *Proc. 40th Annual Convention IPS Japan* (1990), 1167-1168.
5. Hamada, M. et al. Simulator for a Parallel Object-Oriented Total Architecture A-NET, *Proc. 42th Annual Convention IPS Japan*, 1H-8 (1991).
6. Horwat, W. *Concurrent Smalltalk on the Message Driven Processor*, MIT Master's Thesis in Electrical Engineering and Computer Science (1989).
7. Macdowell, E. C. and Helmbold, P. D. Debugging Concurrent Programs, *ACM Comput. Surv.*, **21**, 4 (1989), 593-622.

8. PIERCE, P. The NX/2 Operating System, *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications* (1988), 384-390.
9. SATO, H. et al. Resource Management of PIMOS (in Japanese), *Trans. IPS Japan*, **30**, 12 (1989), 1646-1655.
10. SEITZ, C. L. The Architecture and Programming of the Ametek Seres 2010 Multicomputer, *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications* (1988), 33-36.
11. *NCUBE2 6400 Series Supercomputer, Technical Overview* (in Japanese), Sumisho Electoric System Co., p. 17 (1990).
12. UNGAR, D. Generation Scavenging: A Non-distruptive High Performance Storage Reclamation Algorithm, *ACM SIGPLAN Notice*, **19**, 5 (1984), 157-167.
13. YONEZAWA, A. *ABCL: An Object-Oriented Concurrent System—Theory, Language, Programming, Implementation and Application—*, The MIT Press (1989).
14. YOSHINAGA, T. and BABA, T. Message Reception Mechanism of the A-NET Local Operating System (in Japanese), *OS Workshop report, IPS Japan* 48-5 (1990).
15. YOSHINAGA, T. and BABA, T. A Parallel Object-Oriented Language A-NETL and Its Programming Environment, *Proc. COMPSAC '91* (1991), 459-464.
16. YOSHINAGA, T. et al. A Node Processor for the A-NET Multicomputer and Its Execution Scheme (in Japanese), *Proc. Joint Symp. on Parallel Processing '91* (1991), 189-196.

(Received May 7, 1991; revised August 26, 1991)