Object Oriented Load Distribution in DinnerBell

S. Kono, K. Tatsukawa,† T. Aoyagi,‡ Y. Kohda,§ H. Tanaka*

Sony Computer Science Laboratory Inc.
E-mail: kono@csl.sony.co.jp
3-14-13, Higashi-Gotanda, Shinagawa-ku, Tokyo 141, Japan
†NEC Corporation
‡Department of Computer Science and Information Mathematics,
The University of Electro-Communications
§ FUJITSU LIMITED IIAS-SIS
* Department of Electrical Engineering, The University of Tokyo

Abstract

Here we describe an object-oriented language based on fine-grained parallelism. This language, called DinnerBell, is based on *single-assignment rule* and *data driven execution*. We use *messageJoin* as a synchronization mechanism. DinnerBell is implemented using *micro-message* technique. DinnerBell also uses a new method called *object-oriented load distribution*. This method works like a macro data flow, however, it works automatically and is much more controllable. The simulation results of this method are also examined.

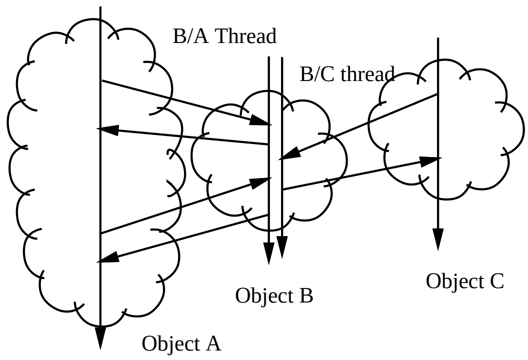

Fig. 1: Multi-threaded Objects Interaction.

1 Fine-Grained Parallel Language: DinnerBell

DinnerBell [kohda84] is a fine-grained parallel object-oriented programming language. It is designed to achieve high software productivity from its object-oriented feature and high execution performance from its fine-grained parallelism.

Several parallel object-oriented languages have been proposed [Yonezawa87, Yokote87, Ishikawa87]. However, they are based on coarse-grained parallelism and their objects are single-threaded. This reduce parallelism and controllability of objects. As in Fig1, some of them introduced multiple threads in rather ad hoc way, with message priorities and express messages.

The concept of dataflow is considered to be effective for introducing fine-grained (and high) parallelism within an object. Dataflow machines have been extensively investigated in the past decade. Hardware prototypes have also been operational (e.g., [Shimada87]). Several programming languages for dataflow machines (e.g., [Ashcroft86]) and fine-grained parallel programming languages (e.g., [Ueda85, Steele87] have also been developed. Some of them employ a combination of dataflow concepts and object-oriented concepts. Such combination has also been tried in some macro dataflow languages [Grimshaw87, Kaiser87]. These languages allow multiple activities within objects. DinnerBell's approach is unique in that it provides fine-grained execution.

The key concepts in DinnerBell are (1) the single-assignment rule which enables parallel execution within an object and (2) messageJoin which creates a consistent state in fine-grained parallel execution. Our language is a pure message-passing based language with synchronization primitive, like [Ward80]. The major modification is the addition of the synchronization mechanism messageJoin. The approach of fine-grained parallelism used in DinnerBell also appeared in [Zhong87] which has Dataflow constructors such as loop.

In fine-grained parallel execution, we must achieve low communication overhead. DinnerBell uses a new approach for load distribution in multi-processors namely *object-oriented load distribution* (hereafter called OO-distribution). This paper examines the simulation results of this method.

0

1.1 Fine-Grained Parallelism

Our approach is to remove unnecessary side-effects. In DinnerBell all variables have single-assignment nature, i.e. they are pure and free of side-effects. A a dataflow graph is constructed dynamically, during execution. In order to express process communication we need some kind of non-determinism or side effects. Necessary side effects are introduced by a non-deterministic technique called **messageJoin**. Although many languages use **guard** as a synchronization primitive, our approach is different. The advantage of our approach is that non-determinism is separated from conditional statements. States and side-effects can be implemented by the messageJoin.

DinnerBell =
single-assignment variable (pure part) +
messageJoin (side-effect part)

Usually merge or serialization is used in concurrent object-oriented languages [Yoshida88, Agha87], but they restrict objects to be single-threaded. Of course, the pure part is easily executed in fine-grained parallelism.

class Add [
inc:X□⇑ #1 ret: (X +:1)
]

Fig. 2: Simple Program.

Fig. 2 shows a simple program written in DinnerBell. The method **inc:** in class **Add** increments the argument **X** and returns the result to the sender of the message. ⇑ states for the sender of the message. A square bracket separates the message pattern from the method's body.

ret: Z ⇐ Add inc:Y

The **ret:** keyword can be omitted because a syntax sugar that simulates assingment is provided (see Fig.3). But actually, there is no assignment because there is no change of values.

Fig. 3 shows the dataflow graph generated by the message passing program of Fig. 2. Since **+:** is a primitive message, it corresponds to a plus node in the dataflow graph.

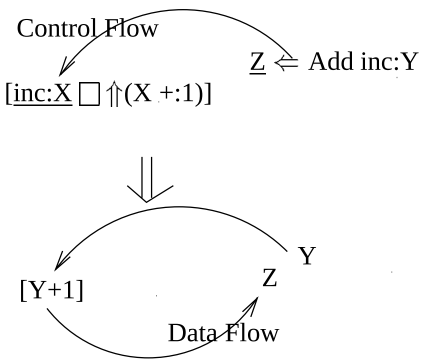

Fig. 3: Dataflow Graph.

1.2 Non-Determinism and State Implemented with MessageJoin

Here we show a usage of the messageJoin technique. Fig. 4 is a simple program that has state. The class Register has three methods:

read!•content: accepts read-requests and returns the content,

write:•content: accepts a write value and changes the content,

new: creates an instance of this class with new content.

There is no distinction between class methods and instance methods in DinnerBell. The first message sent to a class always causes instance creation. The **new:** method creates an instance and then sends a message **content:** to **self**. The default destination of message passing is **self**. Message-passing to **self**, a programming style of Smalltalk, is like a function call within a class.

class Register [
new:X □ content:X
read!•content:X □ ⇑ #1 ret: X.content:X
write:New•content:X □ ⇑ #1 ret: X. content:New
]

Fig. 4: Register.

The second and third methods use the messageJoin mechanism. If both the messages **read!** (unary message) and **content:** arrive at the object, the second method **read!•content:** fires. When several corresponding messages are available, one message is selected non-deterministically. For example, if many **read!** messages and many **content:** messages arrive at the object, one **read!** message and one **content:** message are selected, and these messages fire.

In **Register**, object's state is represented by method variables. Sending the message **content:** changes the state of the object. The first method, **new:**, decides the initial state of the object. Method **read!•content:** returns the current state to the sender of the message **read!** (the sender of the first message is ⇑ #1), and sends the message **content:** with the same state to **self**. ⇑ #1 and self are both pseudo variables. In DinnerBell, the default destination of a message sending is **self**, so we can omit the **self** pseudo variable here. ⇑ #1 denotes the sender of the first of joined messages, and ⇑ #2 denotes the second. The method **write:•content:** sends a **content:** message with a new state to **self**. This causes change of the state.

This simple Register object is an example of transaction in DinnerBell. Here is a possible usage of **Register**. First a register object is created. Then two transaction access the register. Since there are no serial executions in DinnerBell object, the order of two transaction is non-deterministic. Value of the variable **X** can be 0 or 1.

R ⇐ Register new:0.
X ⇐ R read!.
Y ⇐ R write: (Y +: 1)

In Fig. 5, objects first interact by messageJoin, then communicate each other by dataflow and pure message.

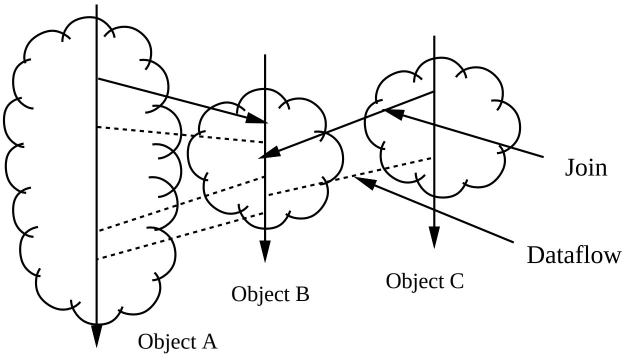

Fig. 5: Dataflow Objects Interaction.

1.3 Examples

We have shown three simple examples of the DinnerBell program. These examples are used in the simulation of the object-oriented load distribution.

1.3.1 Sum

The Sum program in Fig. 6 calculates the sum of a sequence of numbers. The calculation is based on applying divide-and-conquer method to a binary tree generated previously. Sum from 0 to: 1000 starts the program.

```
class Sum [
 from: X to: Y □
     (X <>: Y)
         yes:(□⇑ (
             (Sum from: X to: ((X +: Y) /: 2)) +:
             (Sum from: (((X +: Y) /: 2) +: 1) to: Y)))
         no:(□⇑ X)
]
```

Fig. 6: Sum.

1.3.2 Quick Sort

The quick sort in Fig. 7 and Fig. 8 shows how to use simple list nodes. A prefix ~ means an **itBlock variable**, an instance variable in DinnerBell. We sometimes call this ‘‘itVar’’ for short. Other variables are method variables(‘‘mVar’’ for short). The class **NULL** and **Node** represent lisp cells in a non-destructive data structure. The class **Split** first splits a list into two pieces, then computes the recursive procedure **qs**. Three classes **Generate**, **split** and **qs** work together in pipeline. Notice that an if-then-else structure is constructed as a nested class definition. Each block has a default message pattern; **eval!**, like in Smalltalk. qs qs:100 generates a list of 100 random numbers and sorts it.

```
class NULL [
   isNull! □ ⇑ TRUE ;
   append: Tail□⇑ Tail ;
   print: Dest
]

class Node [
   isNull! □⇑ FALSE;
   cons: ~Car and: ~Cdr ;
   car! □⇑ ~Car ;
   cdr! □⇑ ~Cdr ;
   append:Tail□
      ⇑ (Node cons: ~Car and: (~Cdr append: Tail )) ;
   print: Dest □ ~Cdr print: ((Dest write:~Car) write: '' '')
]
```

Fig. 7: List Node.

1.3.3 N-Queen

The class **Q** in Fig. 9 solves the N-Queen puzzle. This rather complicated example is used in evaluation of OO-distribution. Q run: 6 starts six queen puzzle.

2 Execution Model - micro-message

To execute our language, we use a set of communication units called micro-messages. A micro-message is a unit smaller than normal message passing. There are three types of micro-messages, as shown in table 1.

argument passing unit,

Destination	Selector:	Argument	Reply	ReplySelf

dereference request,

Dereference	Destination	Source

reply.

Reply Value	Destination

Table 1: Three types of micro messages.

The DinnerBell compiler decomposes the source program into micro-message passing. Each (normal) method definition is compiled into micro method definitions (Fig. 10), which have only one argument. Synchronization constructs are also compiled in the same way, using a special side-effect object called $**manager**. This is the side-effect part of DinnerBell.

The execution of a micro-message is very simple. Assume there are several micro-message sendings. They are enqueued in each processor element (hereafter called PE) or are transmitted from another PE. A PE picks up one of these messages and

```
class Split [
 split: ~S of: List □
        ⤊ high: (high: List). ⤊ low: (low: List) ;
 high: List □
     (List isNull!)
      no: ( □ ((List car!) >: ~S )
         yes: ( □ ⤊ (Node cons: (List car!) and:
             (high: (List cdr!))))
         no: ( □ ⤊ (high: (List cdr!))))
      yes: ( □ ⤊ NULL) ;
 low: List □
     (List isNull!)
       no: ( □ ((List car!) <=: ~S )
         yes: ( □ ⤊ (Node cons: (List car!) and:
             (low: (List cdr!))))
         no: ( □ ⤊ (low: (List cdr!))))
      yes: ( □ ⤊ NULL)
]

class qs [
 sort: List □
     (List isNull!)
       no: ( □
         high: H low: L ⇐ Split split: (List car!) of: (List cdr!).
         ⤊ ((qs sort: L) append:
          (Node cons: (List car!) and: (qs sort: H))))
       yes: ( □ ⤊ NULL)

 qs: N □
         list1 ⇐ Generator to: N using: 103. list2 ⇐ qs sort: list1.
]
```

Fig. 8: Quick Sort.

```
class NULL [
 cantake: aQueen □⤊ FALSE ;
 print: N □ StdErr write: N nl!
]

class Queen [
 x: ~X y: ~Y ;
 link: ~Next ;
 x! □⤊ ~X ;
 y!□⤊ ~Y ;
 isNull! □⤊ FALSE ;
 cantake: aQueen □
    (((~Y =: (aQueen y!))
         or: ((~X +: ~Y) =: ((aQueen x!) +: (aQueen y!))))
         or: ((~X −: ~Y) =: ((aQueen x!) −: (aQueen y!))))
       yes: ( □ ⤊ TRUE)
       no: ( □ ⤊ (~Next cantake: aQueen)) ;
 print: N □ ~Next print: ((N *: 10) +: ~Y)
]

class Q [
 check: ~previousQueens at: ~N till: ~limit □
    (~N =: ~limit)
     yes: ( □ ~previousQueens print:0 ) no: ( □ y: 0) ;
  y: M □
    (M =: ~limit) no:( □ y: (M +: 1) .
    aQueen ⇐ Queen x: ~N y: M.
    aQueen link: ~previousQueens.
    (~previousQueens cantake: aQueen)
       no: ( □ Q check: aQueen at: (~N +: 1) till: ~limit)) ;
  run: ~limit □
    check: NULL at: 0 till: ~limit
]
```

Fig. 9: N-Queen.

searches for the appropriate method definition. If it is a primitive message passing, it is executed. Otherwise it prepares for the necessary local work area (method variable) and instance variable (itBlock variable). Sometimes an object work area is inherited from its parent object. The size of the instance variable is determined by the DinnerBell compiler and described in a micro method definition. These work areas are allocated in this PE, and are not transferred to another PE. Then the PE enqueues the body of the method, i.e. the set of micro-message sendings. Some or all of them are transmitted to another PE.

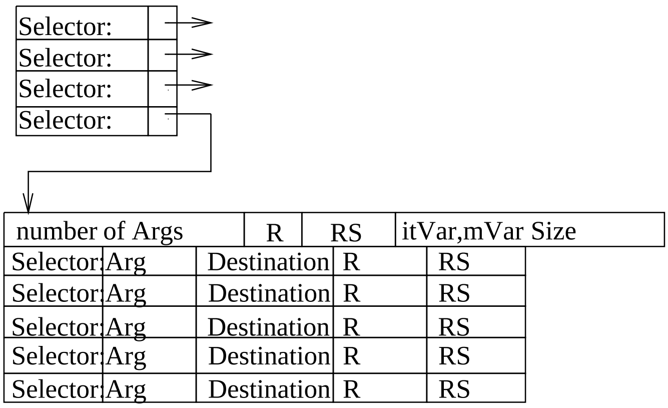

Fig. 10: Micro method definition.

An object in this execution scheme is represented in Fig. 11. The object has its own class id and two data arrays or environments succeeding its outer class. They are called mVar and itVar, respectively.

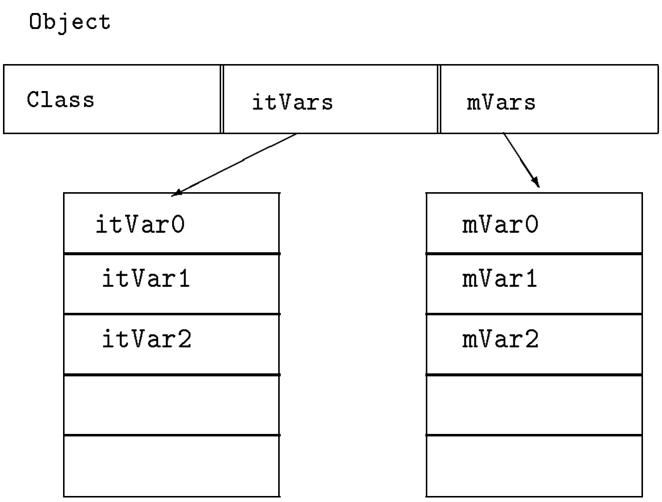

Fig. 11: DinnerBell Object Representation.

The advantages of this execution model are,

1. Fixed-size token fits to the current processor communication.
2. Parallelism is gained in argument passing.

However there are several disadvantages, such as,

1. An argument passing becomes expensive without some hardware support.
2. The execution sequence becomes independent of program sources.
3. Some of gained parallelism is rather imaginary.

We do not think this is the best implementation of DinnerBell. An implementation by a variable-size normal message is also appropriate to DinnerBell, and OO-distribution can also be applied. There are several languages which accept arguments of variable length. Our micro-message is designed to simulate a dataflow token and not to use variable-length arguments. In other words, our execution model is an extension of dataflow. Compared with another dataflow execution of object-oriented language [Zhong87], our execution model is based on a much simpler concept. For example, our system does not need any lock syntax. Moreover, assuming single-assignment rules and recursion-style implementation of states, DinnerBell is more radical than other dataflow object-oriented languages (see [Grimshaw87, Kaiser87]). Using these simple and attractive features we have achieved an effective parallel implementation, as shown in the next sections.

3 Object-Oriented Load Distribution

In this section, we propose an object-oriented load distribution technique. This method mostly uses the object-oriented nature of DinnerBell. Load distribution is important in object-oriented languages because usually an object contains large amount of data that cannot be moved easily from one PE to another PE. We emphasize that this load distribution technique can be generalized for any fine-grained parallel MIMD language, because all have some object-oriented characteristics.

In DinnerBell the destination of a message has a type. This is represented by a tag in this implementation. The types of destination are:

Class Creating new object,

Variable Unbound variable in some PE,

Object May have ItVar and mVar array as shown in Fig.11,

Local Class BLOCK or ITERATION, pseudo variable for a nested object,

Assignment Local class without message passings.

Primitives Such as ARRAY, STRING, FLOAT, and INT.

Load distribution is determined by this classification.

3.1 Three Methods of Load Distribution

Here we show three load distribution algorithms. Each PE picks up one micro-message and decides its destination using these algorithms with no global information about load distribution. The first method (Fig. 12) is only for reference. It uses rotated transmission of micro-messages. Since the number of micro messages cannot be estimated, we can consider this method as a random distribution. The outline of the algorithm is shown in a simplified C language. In this method we may have maximum parallelism, however it produces many dereference/reply messages, consequently causing communication bottleneck.

```
Random() {
    if ((i = next_pe++)>max_pe)
        next_pe = 0;
    return i;
}
```

Fig. 12: Random Load Distribution.

In this random distribution, the object structures are fully distributed in the PEs. Temporary variables and permanent variables of an object can reside in different PEs. This makes possible to execute a program totally in parallel; however, the resulting numerous interprocessor linkages generate many dereference messages (see Fig. 13, iv means **itVars** and mv means **mVars** here).

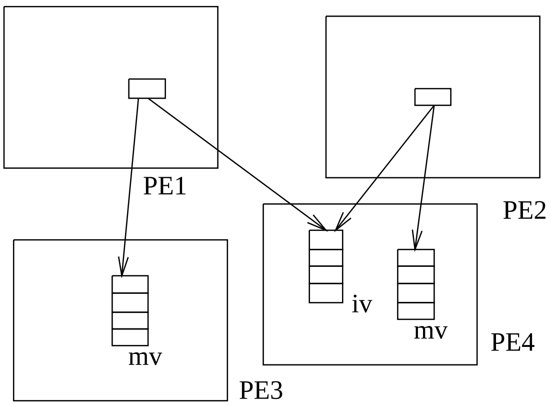

Fig. 13: Object Location in Random Load Distribution.

Another distribution algorithm is shown in Fig. 14. The previous algorithm ignores the location of the existing destination object. It is necessary to reduce the dereference/reply micro-messages. This distribution traces an object's reference, so we call this a reference distribution. This algorithm is rather complicated, because if the destination is an object we have to determine its actual location:

1. If, as in the first case, the destination is a variable, the processor sends this micro-message to the PE to which the variable is allocated. If, as in the second case, the destination is an object, a processor checks whether or not this message is an assignment.

2. A message passing is an assignment only if it does not produce any other messages or side-effects. (This is checked by the compiler). An assignment message is treated only in the destination PE.

3. Otherwise, if the object is local (i.e., a nested object or a class made by compilation of micro-message), the processor sends this message to the PE of the method variable array. This is because local class is a part of its parent object, and the probability of sending a message to an object that has the same method variable environment is high.

4. If the object is a normal object, the micro-message is sent to the PE to which the object is allocated.

5. Otherwise, if message is sent to a class or another primitive, the sending is done randomly.

In this distribution, each message passing is executed in the location of its permanent variable. However, some of the primitive message passing are exported with their temporary variables. The major effect of this distribution is reduction of dereference messages (see Fig.15).

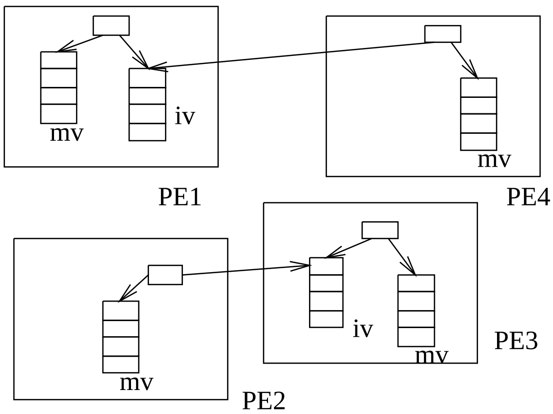

Fig. 15: Object Locations in Distribution for Reference.

The last distribution method, called object-oriented load distribution (OO-distribution), is based on modifications of reference distribution. In this case, however, only two types of message are going outside: (1) the object creation message and (2) message whose destination is located outside.

In this distribution method, we expect to reduce unnecessary message passings, with the result that there will be no parallel execution in an object. All the objects are closed in its PEs (see Fig. 17). The distribution is made only at object creation. None of the primitive message passings are exported. In order to get parallelism, it is necessary to create separate objects, and message passing are then executed in parallel according to their distribution. In other words, this method only supports data

```
Reference(message) {
    pe = pe_of(message.destination);
    switch (type_of(message.destination)) {
    case Variable:
        return pe;
    case Object:
        if (sending_for_assignment(message)) return pe;
        if (sending_to_local_class(message.destination))
            if (have_methodvar(message.destination))
                return pe_of_methodvar(message.destination);
            else
                return self_pe;
        else
            if (have_itvar(message.destination))
                return pe_of_itvar(message.destination);
            else
                return self_pe;
    default:
        return Random();
}
```

Fig. 14: Distribution for Reference.

```
Object(message) {
    switch (type_of(message.destination))
    case Class:
        return Random();
    case Object:
    case Variable:
        return Reference(message);
    default:
        return self_pe;
}
```

Fig. 16: Object-oriented Load Distribution.

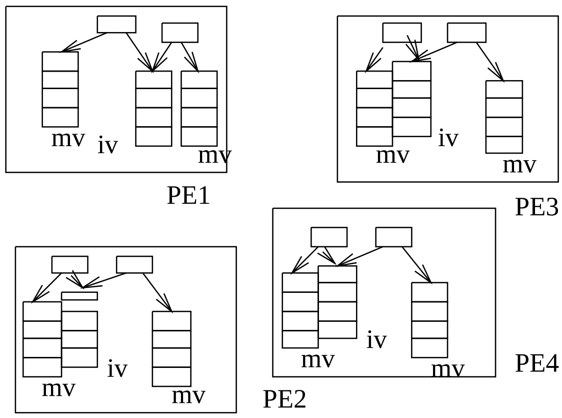

Fig. 17: Object Location on object-oriented load distribution.

parallel execution. This distribution will be effective in the situation of a low-calculation cost and high-communication cost.

In the above three distribution methods, we use neither global information nor additional information except the program source. This makes our language simpler and more controllable. A programmer can design the distribution strategy by designing the object structure in the program.

4 Implementation and Simulation

In this section we show the simulation results of our language and distribution method. Simulation of the parallel computer architecture is very difficult. The result of the simulation does not give actual speed, but only a qualitative result.

4.1 Implementation Model

In this simulation, we use a simple multi-processor model as shown in Fig.19. Each processor has its own local memory and communicates with each other only by passing messages through the network. The network transmission unit and receiving unit work in parallel with micro-message execution unit. The network assumption is also simple. We assumed the network to be flat, that is, with no locality. A multistage network is an example of such a network. This architecture assumption is general, as in [Dally87].

Micro-message execution time is based on a micro-message

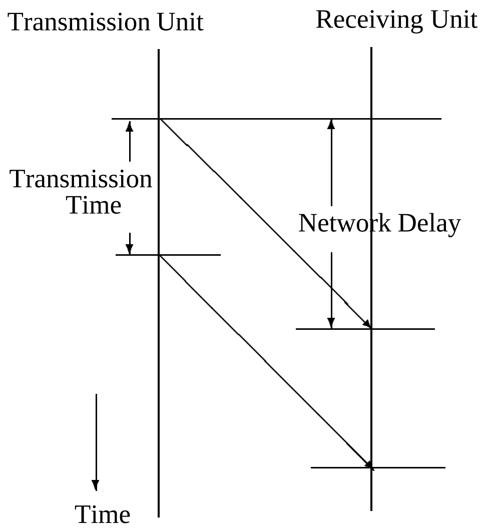

Fig. 18: Time Chart of Simulation.

interpreter on MC68000 (0.5Mips) micro-processor and a multi-stage network LSI. Our assumptions are (1) one processor takes 1.39 ms to execute one micro message, (2) 350 μs to suspend one micro-message, (3) network delay 10 μs, (4) network transmission time 20 μs for one micro-message. For the 6-Queen program, our system (16 parallel processors) has almost the same speed compared with the assembler program in Vax11-730 (0.2 Mips) with these assumptions. These assumptions are realistic enough for estimation purposes. Transmission time determines throughput of the networks. On the other hand, network delay means latency only. These timings are shown in the simulation time chart as (Fig.18). In the multi-staging network, we can get a high throughput with a rather large delay. In data-driven architecture like ours, as shown in a later section, the network delay is not an important factor, so the multi-staging network will be effective in our system.

We used three different kinds of test program. (1) Sum: A computation of the sum of 1 to 1000 by divide-and-conquer. This program has well separated parallelism, and shows the maximum capability of our system. (2) Quick Sort: Sorting of 100 random numbers in a list. This program is a stream-oriented program. Though it does not have much parallelism, we can see the effect of pipeline execution from this example. (3) 6-Queen: Finding all solutions of the 6-Queen puzzle. This is the most complicated program in this simulation. It has some parallelism and communication problems in it since it shares rather big objects.

Fig. 20,21 and 22 show processor utilization. Utilization is an average of effective micro-message execution units during whole computation. If there is no overhead in parallel computation, the average is the same as the number of processors, that is, the graph is linear. The linearity of average almost corresponds to speed-up in parallel execution. If parallelism in a program are all used, the average is independent to the number of processors. It is a constant. The saturation in this graph means saturation of speeding up. These graphs show real parallelism in the problem.

In this simulation, the cost of communication is rather low, so that we can get high parallelism in random distribution. All three types of distribution strategies show the same utilization, and it reflects the parallelism of the program. This means that we do not loose parallelism in the problem using OO-distribution.

Total utilization is 40 $\sim$ 50%, which is not so high. This is partially because of the tree structure of the program, and partially because our system does not search empty processors. In an actual system, some mechanism such as load balancing information will be necessary, but in this simulation the effect of the OO-Distribution is important. So we omit this information. Such a mechanism will make the utilization higher, but will not help to solve the communication bottle-neck.

In spite of poor utilization, the increased speed of this system is good. Even with 64 processors, the speed-up is not saturated except for the relatively low-parallelism program Quick Sort.

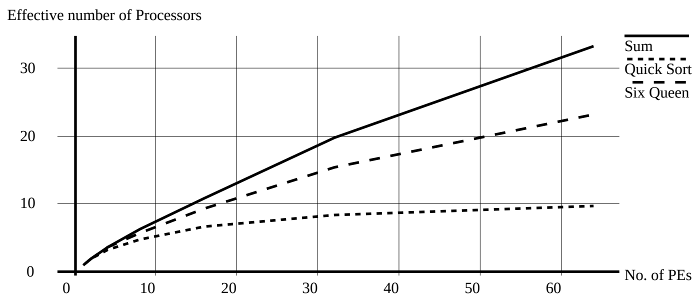

Fig. 20: Number of PE vs. Mean Effective PE: random distribution.

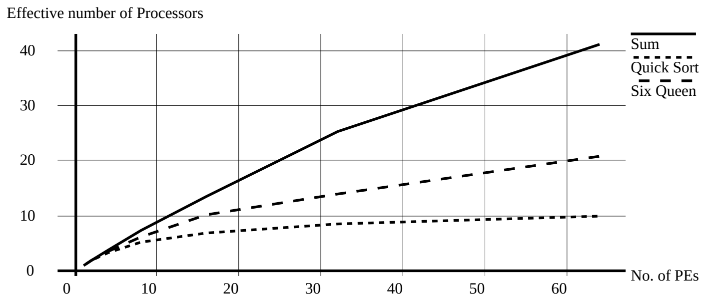

Fig. 21: Number of PE vs. Mean Effective PE: reference distribution.

4.2 Evaluation of Network Delay

Network delay is not so serious in a highly pipelined system. In other words, we can see the effect of network delay as a test for the effect of pipeline execution (See Fig. 23,24,25). Among the three test programs, the Quick Sort program is the most pipelined program. For the other two, the three distribution strategies give

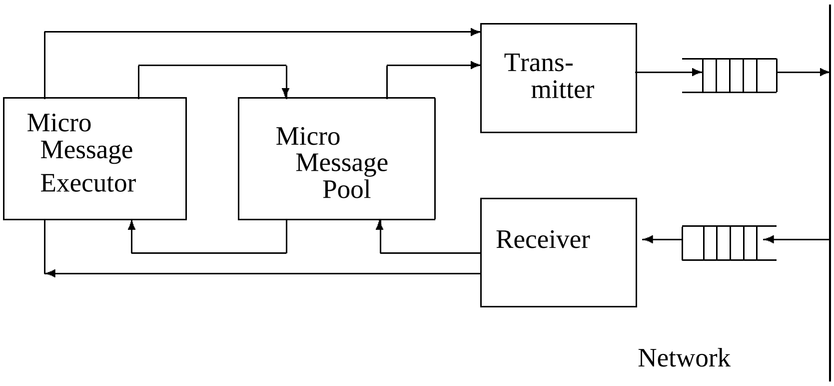

Fig. 19: A simple multi-processor model.

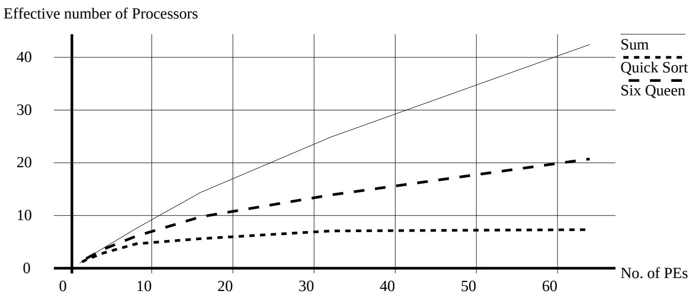

Fig. 22: Number of PE vs. Mean Effective PE: OO distribution.

us the same result. However, in the Quick Sort program, only OO-distribution shows a flat curve. The distribution strategies other than the OO-distribution have some redundant transmissions. This causes disturbance of the pipeline effect.

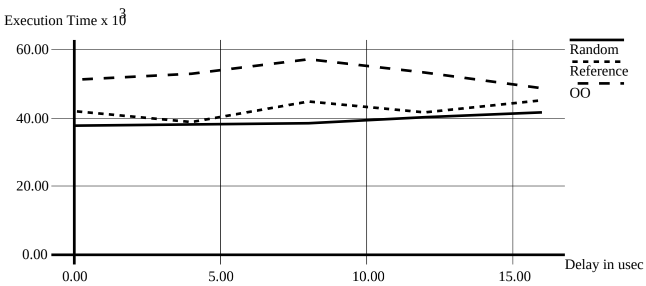

Fig. 23: Network Delay vs. Execution time: Sum.

4.3 Evaluation of Network Throughput

The good results in Fig.20 depend upon high throughput communication. What about a low throughput network? In this case we expect a linear dependency between execution time and network transmission time, because in most multi-processor systems there are communication problems. This occurs especially in large systems such as a 64-processor system. However, our simulation results give us a surprise (See Fig. 26,27,28). In the OO-distribution we can see almost flat curves. This means that, in

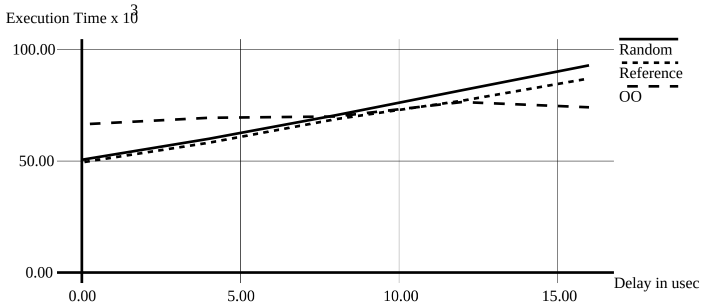

Fig. 24: Network Delay vs. Execution time: Quick Sort.

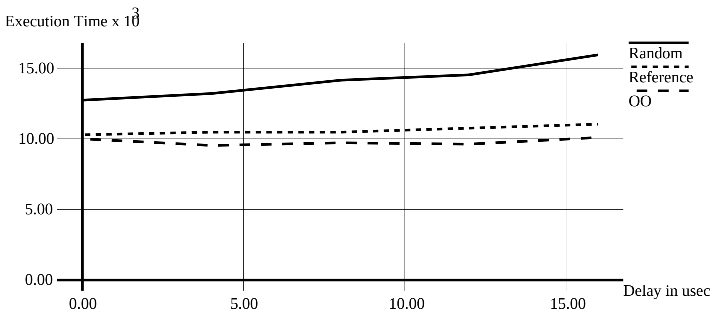

Fig. 25: Network Delay vs. Execution time: 6-Queen.

our system where network transmission time is $40\mu sec$, we have no communication bottle-neck as shown in Fig. 29. These results are of course problem dependent; nevertheless, three different types of this test program give us the same result. Our distribution strategy can extract enough parallelism from the structure of the program. The results reflect the nature of the parallel execution structure of the test programs.

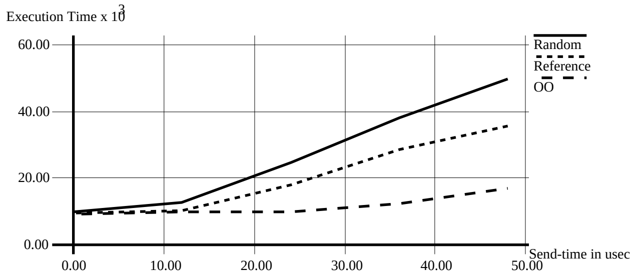

Fig. 26: Network Transmission time vs. Execution time:Sum.

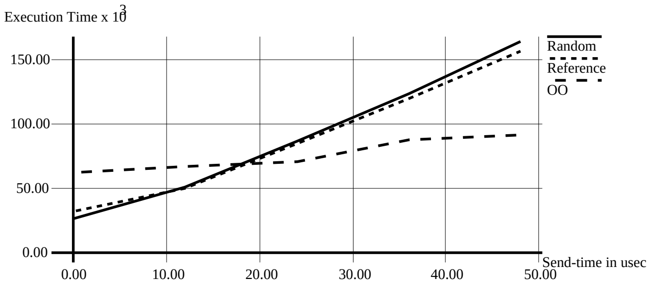

Fig. 27: Network Transmission time vs. Execution time:Quick Sort.

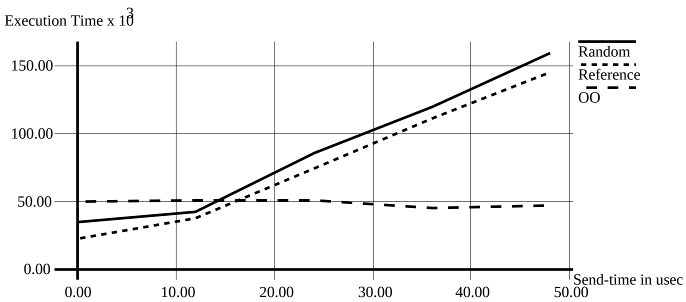

Fig. 28: Network Transmission time vs. Execution time:6-Queen.

5 Conclusion

In this paper we introduced a new parallel object-oriented language. DinnerBell and an object-oriented load distribution strategy. We consider this language to be an object-oriented one not only because of its class-method syntax but also because of its object-oriented distribution strategy. Why does the OO-distribution give us a good result? There is a simple reason. If we have separate data on separate processors, we can work on these

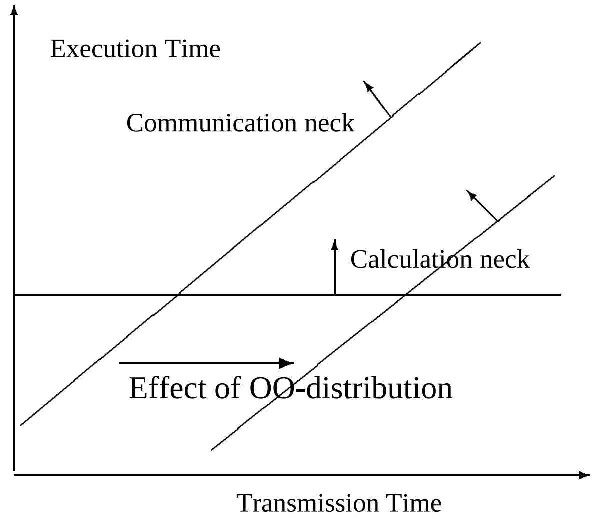

Fig. 29: Bottle neck in object-oriented Load-Distribution.

data in parallel. OO-distribution will map each object onto separate processors during execution and messages that depend on the object will not be sent outside. Therefore, the processor can work in parallel, in a low communication frequency between objects. OO-distribution makes it possible to minimize the communication bottle neck. The actual effect of this method is an extraction of data parallelism from our fine-grained parallel programming language. OO-distribution can be applied to other fine-grained parallel architectures. If we can construct well-designed parallel computer architecture for the micro-message system, this will be the best parallel architecture.

References

[Agha87] G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[Ashcroft86] E. A. Ashcroft. Dataflow and eduction: data-driven and demand-driven distributed computation. In *Current Trends in Concurrency, LNCS 224*. Springer-Verlag, 1986.

[Dally87] W.J. Dally, A. Chien L. Chao, S. Hassoun, W. Horwat andJ. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *14th Comp. Arch. Conf. Procs.*, pp. 189--196. IEEE, 1987.

[Grimshaw87] A.S. Grimshaw and J.W.S. Liu. Mentat: An Object-Oriented Macro Data Flow System. In *OOPSLA 87*, pp. 35--47. ACM, 1987.

[Ishikawa87] Y. Ishikawa and M. Tokoro. Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation. In *Object-*

Oriented Concurrent Programming. MIT Press, 1987.

[Kaiser87] G.E. Kaiser. MELDing Data Flow and Object-Oriented Programming. In *OOPSLA 87*, pp. 254--267. ACM, 1987.

[Steele87] G.L. Steele and W.D. Hillis. Connection Machine LISP: Fine-Grained Parallel Symbolic Processing. In *1986 ACM Conf. on LISP and Functional Programming*, pp. 279--297. ACM, 1987.

[Shimada87] T. Shimada, K. Hiraki, and K. Nishida. Evaluation of a prototype data flow processor of the sigma-1 for scientific computation. In *Proc. of 13th Annu. Symp. on Computer Architecture*, pp. 267--234. IEEE, 1987.

[Ueda85] Kazunori Ueda. Guarded Horn Clause. Technical Report TR-103, ICOT, Jan. 1985.

[Ward80] S.A. Ward and R.H. Halstead. A syntactic theory of message passing. *J. ACM*, Vol. 27, No. 2,, 1980.

[Yoshida88] Kaoru Yoshida and Takashi Chikayama. A'UM - a stream-based concurrent object oriented language -. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*. ICOT, 1988.

[Yonezawa87] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modeling and programming in an object-oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[Yokote87] Y. Yokote and M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[Zhong87] Y. Zhong and M. Sowa. Towards an Implicitly parallel Object Oriented Language. In *Procs. of Compsac 87*, pp. 481--485, 1987.

[kohda84] Y. Kohda , S. Kaneko , H. Tanaka and T. Moto-oka. An Overview of Parallel Object Oriented Language DinnerBell Technical Report SF-11-3, IPSJ, 1984. (in Japanese)