プログラミング 8－26
（1996. 8. 29）

The Implementation of A-NETL on a Highly Parallel Computer AP1000

Somchai Numprasertchai† Tsutomu Yoshinaga† Takanobu Baba†

並列オブジェクト指向言語 A-NETL は，中粒度の小～大模並列処理を行なうことを目的として設計され，ユーザは A-NET マルチコンピュータ上で効率的な実行が可能である．また，様々な並列・分散コンピュータにも A-NETL が実装されつつある．

本稿では，並列オブジェクト指向言語 A-NETL の高並列計算機 AP1000 への効率的な実装について述べると共に，実験結果を示す．

The Implementation of A-NETL on a Highly Parallel Computer AP1000

Somchai Numprasertchai† Tsutomu Yoshinaga† Takanobu Baba†

A-NETL (Actors-NETwork Language) is a parallel object-oriented language intended to be used for managing small to massive parallelism with medium grain size. The basic motivation of the A-NETL design is to allow the users to describe large parallel programs and execute them efficiently on a highly parallel machine, named the A-NET multicomputer. A-NETL is also being implemented on various parallel and distributed computers. This paper describes the efficient implementation of the parallel object-oriented language, A-NETL on a stock multicomputer, AP1000. Some preliminary results are also included.

Keywords : parallel object-oriented language, message passing, language implementation

1 Introduction

Over the years both the scientific/engineering and the commercial communities have placed growing demands on computer hardware and software. This has led to dramatic improvement in computer architecture to increase processing power and cut program execution time. In addition a lot of problems have parallel behavior inherent in their algorithms. Therefore high performance machines and parallel programming languages are necessary. Object-oriented languages which are recognized as a promising approach to parallel programming are considered. There are many object-oriented programming languages that have been implemented on several kind of multicomputers, such as ABCL/onAP1000.[2]

We have developed a parallel object-oriented total architecture system, A-NET. This system consists of the A-NETL, a local operating system and the A-NET multicomputer. We have developed language processors for a variety of environment such as A-NETL on a stock machine, AP1000 machine, and A-NETL on a cluster of workstations using PVM[5].

The purpose of this paper is to demonstrate an efficient implementation of an AP1000 translator which converts A-NETL programs to C language programs. We begin by describing the overview of A-NETL and AP1000 system in section 2. Section 3 presents the implementation of the AP1000 translator and optimization method. Section 4 shows the experimental results.

† 宇都宮大学工学部
Faculty of Engineering, Utsunomiya University

2 A-NETL and AP1000

2.1 Overview of A-NETL

This section briefly describes our language, A-NETL. A more comprehensive description of the language A-NETL can be found in [3,4,7]. A-NETL provides a tool for defining a large number of objects both statically and dynamically. In static definition, indexed objects are groups of a large number of similar objects. They can be defined by attaching the number of required objects to object name as follows.

$$objectName[\text{NumberOfObject}]$$

For dynamic creation, a class is defined as a mold object. The class definition is broadcast to all nodes before execution. This is useful for programs that need to reconfigure their structure dynamically according to input data size and runtime parameters.

A-NETL uses message passing to communicate between objects. It has three types of message passing, past, now and future[2] and their multicast versions.

2.2 AP1000 multicomputer

The AP1000 is a highly parallel computer with distributed memory which is suitable for fine-gain and data parallel programs[6]. Each processor(cell) runs its own program from its local memory and communicates with other cells by passing messages. Messages are sent from the source cell to destination cell though the communication network, Broadcast network(B-net), Torus network (T-net) and Synchronous network (S-net)[1].

In AP1000, there are two types of programs, *Host programs* and *Cell programs*. They are compiled and linked with libraries at the host computer to create execution modules. There are many communication functions supported the AP1000 library. In our implementation, we use *h_send* to send a message from cell to host and *l_asend* to send a message to a specified task identifier(tid) in a one-dimensional absolute cell ID(cid).

3 Implementation

3.1 Configuration of the translator

We selected to translate A-NETL code to C language code. The decision on using the C language was based on maintaining portability on various current and future machines. However, A-NETL and C language syntax and semantics are different. For example, message reception in A-NETL is supported by the operating system in the A-NET machine while we have to operate them in the user mode in C language on AP1000. So we have to implement several functions for things such as polymorphic data, method, message and object management. In addition A-NETL is supported by A-NET machine hardware.

In our implementation, the AP1000 translator is divided into 2 phases. In Phase 1, an A-NETL source file is translated to an intermediate language and in Phase 2, the intermediate language is translated into a C language program. Both phases of the translator are grouped together as the translator driver. The compilation and execution process of A-NETL on AP1000 is shown as **Figure 1**.

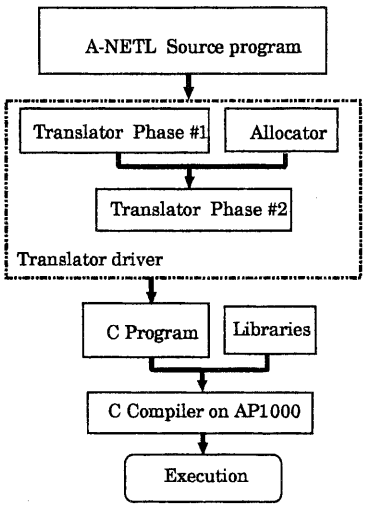

Figure 1. The compilation and execution process of A-NETL on AP1000.

3.2 Compilation of A-NETL

3.2.1 Methods

Methods contain temporary variables, and statements that include several primitive operations such as arithmetic and logical operations. Method execution is supported by A-NET system.

In the C language, we have to execute methods in user program level. We translate a method to a C function and create several functions such as method selector to select the current method to be executed. A-NETL primitive operations are implemented by cell libraries.

3.2.2 Message

We have to create several functions to implement message sending and receiving on AP1000. In the implementation, we need to realize the message send/receive semantics on ready-made message passing libraries. This forces us to define it as a user-mode program without utilizing special support from the underlying OS and architecture.

We designed the object identifier structure as shown in **Figure 2** so that it can be easily mapped to the AP1000 message passing format of 3 parts, *Object_type* contains the object type, *Object_No* is mapped to AP1000 cell identifier (or is equal 0 when it refers to the host program) and *Node_ID* is mapped to the AP1000 task number.

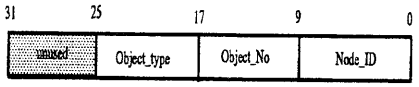

Figure 2. Object identifier structure.

The receive object changes information in the object identifier to the AP1000 format by changing *Node_ID*, *Object_No* and checking the message type (0:return, 1:past, 2:now and 3:future type) in this structure. The message structure is shown in **Figure 3.**

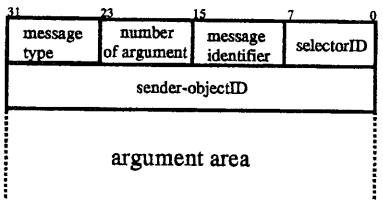

Figure 3. Message structure.

We created functions to map the three types of A-NETL message and their multicast versions to AP1000 message passing functions by using *h_send* for communication between cell and host and *l_asend* for communication among cells.

3.2.3 Indexed object and class

We keep indexed objects in indexed objects table and refer to them by their index numbers. They also refer themselves by using selfIndex.

For dynamic creation, we assign the dynamic number for class and dynamic objects. The dynamic number for a class is zero. Numbers other than zero are assigned to dynamic objects.

A class consists of methods, state variable values for each dynamic object and a list which points to the first dynamic object. Each dynamic object then has pointers which point to its' variable values in the class and the next dynamic object as shown in **Figure 4.**

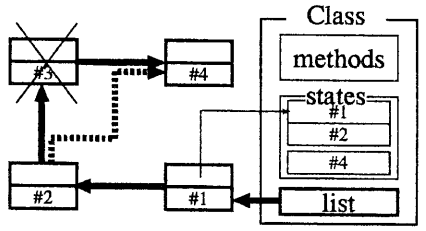

Figure 4. Class operation.

3.3 Code optimization

We have tried to implement the translator to produce a optimized code. However the code from translator is still not as efficient as when the code is hand written.

We therefore hand-optimizing the resultant C code, changing some structure and reducing overhead as follows.

1) Instead of dynamic memory space allocation for things such as buffer, we do it statically.

2) Reduction of the overhead of method and condition control by inlining code.

3) Reduction of the number of steps in message passing.

4) If a context is ready and is not stored in context queue, it is executed directly.

Another important process in code optimization is a TAG treatment. A-NETL implies dynamic data types. This is good for execution on the A-NET multicomputer which has a TAG processor. However, this data type causes a large overhead. In the optimization process, we changed the TAG type to normal types in C, to reduce the overhead and required memory.

3.4 Allocation of A-NETL objects to AP1000 cells

The translator driver is designed to receive the size of cell in X and Y axises for torus topology in AP1000 from user.

We setup the configuration of the AP1000 cells using the $cconfxy(ncelx, ncely)$ function and we use *l_ccreate* to create and activate tasks in cells specified by one-dimensional absolute cell ID. This process is implemented in the *host program*.

One A-NETL object is allocated to one task in one cell when there are enough cells compared with the number of A-NETL objects. Otherwise, multiple A-NETL objects are allocated to one cell with different task identifiers, considering load balance. For example, we allocate indexed objects with the same load to all cells in the first step and then allocate the rest of the objects to cells in another task identifier when the number of cells is less than the number of objects.

3.5 System management

We designed a user interface as a host program. The host program includes a message handler, a message dispatcher and a message interpreter to allow the user to process an input message.

We have also implemented function called an object manager to manage message passing. There are 2 main processes in the object manager, (1) receive a message and create a context and (2) start methods from context.

Figure 5 shows the structure of object manager. The bold line shows the process flow from message transmission to message execution.

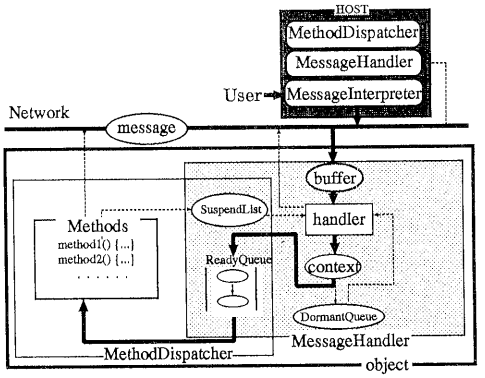

Figure 5. Object Manager.

1)Message Interpreter: The message interpreter receives and interprets input messages from the user.
2)Message Handler: The message handler creates a context and puts it into the ready queue. For multireceive, contexts are placed in the *DormantQueue*. In the case of a return type message, the message is put into the *SuspendList*. If the corresponding context is found, the return message, stored in this context, will be placed in the *ReadyQueue*. In this way, a suspended message is made ready for execution. In the message handler, a buffer is used to reduce the waiting time for arrival messages.
3)Method Dispatcher: The method dispatcher selects a method allocate in the *ReadyQueue* to execute.

4 Experimental Results

4.1 Environment of experimentation

All of the experimental programs are compiled by the gcc compiler with the optimizing option and their execution time is evaluated on **caren**(CAP Runtime Environment) on the AP1000, which consists of 64 cells by using both -NOLSEND option and -LSEND option.

4.2 Ping-Pong benchmark

To determine the efficiency of message passing, the time taken to send a message from a sending cell to a receiving cell and back was measured. We improved the object manager by inserting a

buffer to reduce waiting time for arrival messages and included a message handler to manage each kind of message. We reduced the two way communication time from 614 μs to 310 μs.

We further optimized the code by applying the strategies in 3.3, and found that the two way communication cost was reduced from 310 μs to 223 μs, while it took 180 μs by writing directly in C language.

4.3 Result of example programs

To demonstrate the effectiveness of this implementation, we show the result of some sample programs, Nqueens program(NQ): a search algorithm to place eight queens, Molecular Dynamics(MD): modeling the behavior of molecules based on the dynamics among 512 particles. The experimental results are shown in **Figure 6.**

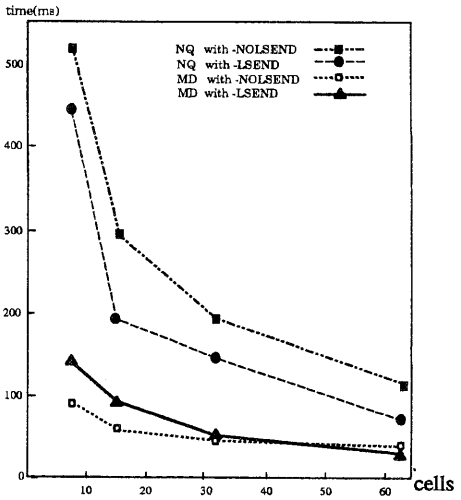

Figure 6. The result of NQ and MD programs.

Note that the resultant AP1000 program has 1 more object than the original A-NETL program. In our experiment, we use 10 objects on 8 cells, 18 objects on 16 cells, 34 objects on 32 cells and 66 objects on 64 cells for NQ program. We used 66 objects on 8, 16, 32 and 66 cells for MD program.

In the early implementation of the NQ program on 8 cells, it took 1053 and 3370 milliseconds for its execution times in -NOLSEND and -LSEND option, because there were heavy load in some cells and light load in other cells as shown in **Figure 7.** And it took only 517 and 444 milliseconds with both options respectively after we reallocated objects to balance load. The load is shown in **Figure 8.** Needless to say, balance of load in cell has effect on the total performance.

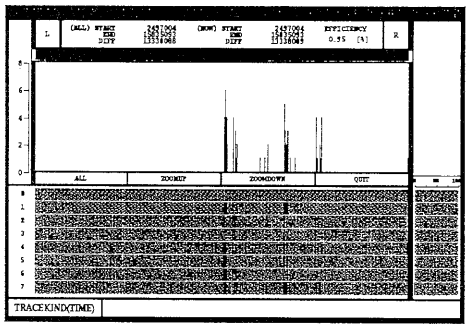

Figure 7. NQ program before balancing load.

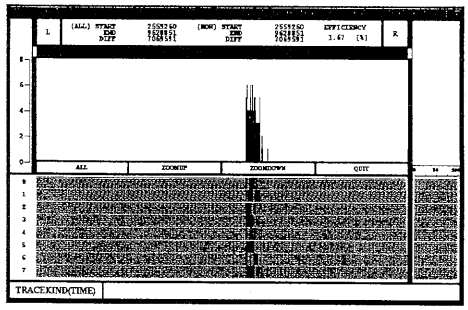

Figure 8. NQ program after balancing load.

We also implemented the NQ program(NQ(2)) with 66 objects on 8, 16,32 and 64 cells with both the NOLSEND option and again with the LSEND option. We compared results in the **Table 1.**

Table 1. NQ in the different number of objects.

Program \ cells		8	16	32	64
NQ(1)	NOLSEND	517 ms	292 ms	193 ms	114 ms
	LSEND	444 ms	196 ms	146 ms	72 ms
	the number of objects	10	18	34	66
NQ(2)	NOLSEND	630 ms	410 ms	208 ms	114 ms
	LSEND	4002 ms	1872 ms	529 ms	72 ms
	the number of objects	66	66	66	66

We have a future plan to process the MD program using a various number of objects, because we found that the number of tasks in each cell has effect on program performance in NQ programs.

5 Conclusion

We have presented the implementation of A-NET language on AP1000 and evaluated it by running sample programs. We show that the execution time is decreased when we increase the number of cells. We clarified that the output performance depends on several factors such as overhead, the balance load of cells, the number of objects.

Our future goal is to encourage widespread use of the A-NET language. We plan to revise the language and its processing system so that it can attain a sufficient speedup in a wide range of applications without requiring special support from an underlying operating system and architecture.

Acknowledgements

This research is supported in part by the Japanese Ministry of Education, Science and Culture under Grant 08680346, the Telecommunications Advancement Foundation, and PDC. The authors would like to thank the Fujitsu Laboratory and University of Tokyo for allowing us to use AP1000 machine and other members of the A-NET project for their helpful comments.

References

[1] H.Ishihata, T.Horie, S.Inano, T.Shimizu and S.Kato: An architecture of highly parallel computer AP1000, In IEEE Pacific Rim Conf. on Communications, Computers and Signal processing pp.13-16,May (1991).

[2] K.Taura, S.Matsuoka and A.Yonezawa: An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers, 4th ACM PPOPP, pp218-228, (1993).

[3] T.Baba and T.Yoshinaga: A-NETL:A Language for Massively Parallel Object-Oriented Computing,Proc. 1995 Programming Models for Massively Parallel Computers(PMMP'95), pp.98-105(1995).

[4] T.Baba, N.Saitoh, T.Furuta, H.Taguchi and T.Yoshinaga:A Declarative Synchronization Mechanism for Parallel Object-Oriented Computation, IEICE Trans. INF. & SYST., Vol.E78-D,No.8,pp.969-981(1995).

[5] T.Furuta. N.Saitoh, A.Tsukikawa, T.Yoshinaga and T.Baba: The Implementation of A-NETL on Workstation Clusters, 1996"Akita Summer United Workshops on Parallel, Distributed, and Cooperative Processing,to appear(1996).

[6] T.Horie, H.Ishihata, T.Shimizu, S.Kato, S.Inano and M. Ikesaka: AP1000 architect and performance of LU decomposition, Proceddings of the 1991 International Conference on Parallel Processing, pp. I-634-635,Aug.1991.

[7] T.Yoshinaga and T.Baba: A Parallel Object-Oriented Language A-NETL and its Programming Environment, Proc. COMPSAC'91 ,pp.459-464(1991).