プログラミング－言語・基礎・実践－ 18－8
（1994. 7. 21）

高並列計算における動的資源管理のための自己反映並列オブジェクト指向言語

増原英彦* 松岡聡† 米澤明憲

東京大学大学院理学系研究科情報科学専攻

データや通信パターンが実行時のみに定まるような、不規則な並列アプリケーションを高並列計算機上で効率よく実行するためには、計算資源の実行時管理ポリシーが重要である。本論文では、様々な動的資源管理システムを柔軟に記述するための自己反映(リフレクティブ)並列オブジェクト指向言語ABCL/R3を提案する。ABCL/R3では、抽象化されたシステムのメタレベルに対する変更・拡張をアプリケーションから隠蔽された形で記述することで、スケジューリング・オブジェクト配置・負荷分散などの資源管理が柔軟に提供できる。本論文ではまた、並列計算機AP1000上に作成したプロトタイプシステムを用いて、予備的な評価を行った。

An Object-Oriented Concurrent Reflective Language for Dynamic Resource Management in Highly Parallel Computing

Hidehiko Masuhara* Satoshi Matsuoka† Akinori Yonezawa

Department of Information Science, the University of Tokyo

Irregular parallel applications, whose data and communication patterns are determined only at run-time, often requires good dynamic resource management (DRM) tailored to the application and/or hardware architecture for efficient execution. To easily provide such DRM system, this paper proposes an object-oriented concurrent reflective language ABCL/R3. In ABCL/R3, various DRM systems including scheduling, object allocation, and load balancing, can be realized by modifying/extending abstracted meta-level of the language in an encapsulated way. This paper also shows preliminary evaluation of the language including a basic cost of reflection and a simple DRM system, developed in a prototype system running on a multicomputer AP1000.

*日本学術振興会特別研究員 (JSPS Research Fellow)
†東京大学工学部計数工学科 (Department of Mathematical Engineering, the University of Tokyo)

1 Introduction

Recently, many concurrent/parallel languages are designed for highly parallel computers. A major target of such languages is irregular (or unstructured) parallel applications, whose data and communication patterns are usually determined only at run-time. Because of the irregularity, policies used in *dynamic resource management* (DRM), such as how a data structure is distributed to processors, greatly affects the performance, and many 'tailored' (i.e., application and/or hardware specific) resource management policies are investigated. Most concurrent languages allows to use meta-level descriptions for DRM. One example is that HPF (High Performance Fortran) has directives for mapping array structures to processors[3]. Another example is that KL-1[11] has primitive directives to specify location and scheduling priority of a newly created process.

Although these fixed or primitive meta-level descriptions are useful in describing simple DRM systems, this approach is not portable. Firstly, with only fixed meta-level descriptions, various resource management policies could not be supported. Since different application algorithms and hardware configurations make different resource management policies to be appropriate, it would be better that the arbitrary resource management policies can be programmed by the application programmer. Sometimes a resource management policy, which can not be directly realized with the fixed meta-level descriptions, may be simulated; but it will be awkwardly described and may have a certain overhead. For example, a depth-first order scheduling on a tree structure, which is easily carried out with a LIFO (last-in first-out) scheduler, may be simulated on a priority based scheduler by giving priority values according to the depth in the tree.

Secondly, with primitive meta-level descriptions, basically, it is possible to construct arbitrary resource management policies. In this approach, however, descriptions for the application itself and for the resource management are intertwisted. As a result, both application and resource management programs are hardly reused. Since suitable resource management policy may change according to the hardware configuration (e.g., the number of processors or the network topology), it would be better that they can be described independently.

This paper proposes an object-oriented concurrent reflective language ABCL/R3 as a platform for describing concurrent applications with tailored resource management systems on highly parallel computers. The language is based on *computational reflection*[8], which is a generalized idea of the computational process that accesses/alters its own computation. In reflective languages, the application programmers can alter the language by accessing/changing abstracted implementation within the language. Moreover, descriptions for the *base-level* (i.e., application level) computation and for the *meta-level* computation can be clearly separated (called *meta-level encapsulation* or *separation of concerns*), it makes both base- and meta-level programs highly re-usable and easily to be reasoned about.

The reflective architecture of ABCL/R3 is based on, and extended from the one of ABCL/R2[6, 5]. We concentrate that it can provide good abstractions of the run-time systems with respect to DRMs, compared to other concurrent/distributed reflective languages[12, 7, 1], which are mainly focused on the language extension. In the following section, the reflective feature of ABCL/R3 is described.

2 ABCL/R3: An Object-Oriented Concurrent Reflective Language

2.1 Overview

Similar to the *Hybrid Group Architecture* (HGA) of ABCL/R2[6], the reflective architecture of ABCL/R3 allows *per-object basis reflection* through the metaobjects, and *group basis reflection* through the shared meta-level objects. It distinguishes itself from the HGA in the notion of nodes (the abstraction of the processor elements on the distributed memory multicomputer) explicitly available at the meta-level. Fig-

ure 1 shows a conceptual sketch of the reflective architecture. The key features are as follows:

- **Notion of nodes** is explicitly available at the meta-level while it is still transparent to the base-level computations.
- Customizable meta-level objects, including **node manager, scheduler,** and **meta-objects**, provide abstractions of the run-time system relevant to dynamic resource management.
- The **chain of executors**, which gives operational interpretation of base-level scripts, provides dynamic and localized language facility extension.
- **Interfaces to low-level systems** are provided so that information on (not customizable) run-time systems—memory usage, for example—can be used in DRM systems, and that facilities such as object migration and timer interrupts can be used.

In the following two subsections, two aspects of the reflective facilities of ABCL/R3—language extension, which alters/extends semantics of the language, and runtime system extensions, which alters/extends behavior of the run-time system—are described, respectively.

2.2 Language Facility Extension with Chains of Executors

Language facility extension is realized by the *Chains of Executors,* which provide interpretations of base-level programs with dynamic and localized customization. Its characteristics are as follows:

Operational Definition: An executor in a chain defines how an interpretation of each base-level expression goes in an operational way. The execution of a base-level expression is represented as a message to executors at meta-level. The message contains all information to evaluate the expression: the expression itself, an environment, a continuation, and so forth. Figure 3 shows a part of the the primary executor's definition. This definition is almost same to the ones in ABCL/R[12] and ABCL/R2[6], which are meta-circular interpreters defined in a continuation passing style(CPS).

Since interpretations are given in an operational way, it is possible to define dynamic interpretations; i.e., the interpretation for an expression can be changed according to a dynamic condition. This ability distinguishes the operational extensions to the compile-time extensions[2].

Scope of Customization: Usually, extension to the language is bounded on a certain group of objects. One example is that an extended message sending mechanism, which is needed for a class, should be bounded on objects in the class. Another example is that the object creation policies are dynamically modified per-node basis according to the load-balancing status; in this case, the modification to the language should be bounded on objects in each node. To do such grouped customization easily, executors are defined for each group: object specific executors, node executors, class executors, and the primary executor. When the user want to modify the language on all members in a group, he re-defines the executor that governs the group.

By default, executors except for the primary one define no specific interpretations; they just delegates requests of interpretation to the next executor in a chain. Figure 2 shows how a script execution is delegated through executors. Consider there are base-level objects foo1 and foo2 in class Foo, and bar in class Bar. Objects foo1 and bar reside at node #1, and object foo2 resides at node #2. In this case, script execution for each object is taken place along the bold arrows in Figure 2. For example, a script execution message for foo1 is delegated in the following order: object executor foo1, node executor #1, class Foo executor, and the primary executor.

Dynamic Replacement: Any executor in a chain except for the primary one can be dynamically replaced. This is accomplished by sending a request to a container object, or by customiz-

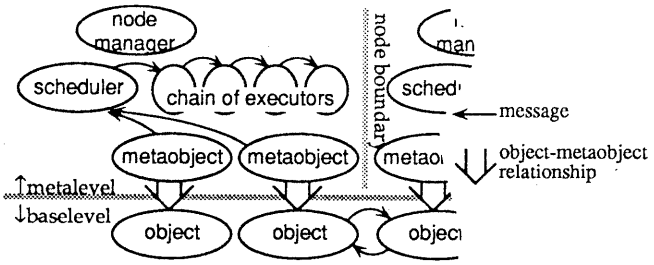

Figure 1: Overview of Distributed Memory Reflective Architecture

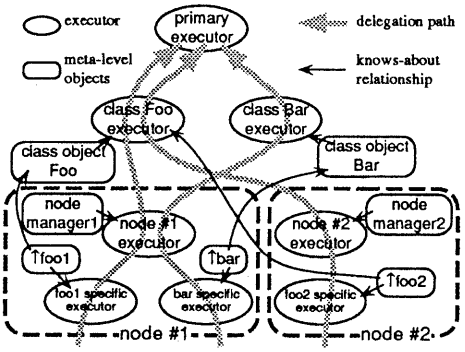

Figure 2: Example of Delegation Path

```
[class primary-executor ()
 (script
  ;; for object creation forms
  (=> [:do [:new class args] env id] @ C
    ;; send to a node manager
    [node-mgr <= [:new class args] @ C])
  ;; for past type messages
  (=> [:do [:send-past target body reply]
           env id] @ C
    ;; send to the metaobject
    [target
      <= [:message body reply [den id]]]
    [C <= nil])
  ;; for variable references
  (=> [:do [:variable name] env id] @ C
    ;; return a value of name in env.
    [C <= (lookup name env)])
        :
 )]
```

Figure 3: Abridged Definition of a Primary Executor

ing a container object to replace its executor dynamically. This facilitates that language extensions in which the functionality of an operation changes dynamically, but not frequently. For example, consider a system that counts number of variable references for specified periods of time. Such a system can be realized by defining an executor which increases a counter whenever a variable is referred. Then the executor is used instead of the original one only for the specified periods. In this realization, the interpretation of variable reference in non-specified periods are defined by the original executor, and no unnecessary overheads are imposed on the variable references.

The reader may have a fear that the operational style extension could not be implemented efficiently. In fact, several reflective systems which have the language extensibility defined in operational style are implemented, and they prove that they are slow in the order of magnitude compared to the corresponding implementations of non-reflective languages[5]. However, more efficient implementation could be possible with sophisticated compilation techniques. Basic ideas for compiling base-level scripts with executors are discussed in [4].

2.3 Runtime System Extension with Meta-level Objects

2.3.1 Node Manager

Node manager is a meta-level object that represents a processor node. Any node manager can

```
[class node-manager ( ... )
 (state [scheduler :=  ... ]
        [executor :=  ... ]
        [mobj-class :=  ... ])
 (script
  (=> [:scheduler] !scheduler)
  ;; replacement of scheduler
  (=> [:set-scheduler new]
      [scheduler <== [:copy-to new]]
      (setq scheduler new))
  (=> [:executor] !executor)
  ;; replacement of node executor
  (=> [:set-executor new-executor]
      (setq executor new-executor))
  (=> [:new class args
        &key mobj-class &rest annon]
   ![den [new mobj-class class args annon]])
)]
```

Figure 4: Definition of a Node Manager

be referred from any object in the system by a pseudo variable or a function call. The forms `node-manager` and (`node-manager-of` n) give references to the node manager where executing object resides and the node manager at node n, respectively. The definition of the default node manager is shown in Figure 4.

The major purpose of a node manager is to hold meta-level objects shared by members in the same node. By default, it holds two meta-level objects: scheduler and node-executor. In addition, arbitrary meta-level objects and information can be held by customized node managers.

The other role of a node manager is local object creation. By default, execution of an object creation form (`[new ...]` form) at the base-level produces a request message to the node manager. Then the node manager decides where the new object should reside; if it is local, the node manager itself creates a metaobject. Otherwise, it selects a remote node, and forwards the request to the node manager at the node.

To construct dynamic resource management systems, user defined node manager is often used to keep track of load information of nodes, exchange load information, decide scheduling/object allocation policies, and so forth.

```
[class scheduler ()
 (state [queue := (make-queue)])
 (script
  (=> [:req-exec thunk]
    (queue-put thunk queue))
  (=> [:schedule-next]
    (let* ((thunk (queue-get queue))
           (context (thunk-context thunk))
           (id (thunk-owner thunk))
           (executor [id <== [:executor]]))
      [executor <= context @ id])))]
```

Figure 5: Definition of Default Scheduler

2.3.2 Scheduler Object

Applications in which amounts of computation greatly vary among objects, or applications whose amount of computation changes according to the order of computation, can be executed more efficiently with tailored scheduling policies. In our architecture, a scheduler object, which is a meta-level object that controls execution order of base-level objects in a node, is the abstraction to define customized scheduling. The scheduler only controls the execution order and time quantum of objects, but it is enough for most dynamic resource management systems.[1]

A default scheduler, whose definition is presented in Figure 5, has a state variable that holds active objects. The variable is an abstraction of an active object queue in a node.

Basically, a scheduler receives an execution request by a message `[:req-exec thunk]`, where `thunk` is data structure holding enough information to execute. The execution is started by sending a context (e.g., an expression, an environment, etc.) to an object specific scheduler. When the system needs an object to be executed, a message `[:schedule-next]` is sent to the scheduler. Then the scheduler picks up a thunk from its scheduling queue, and schedules it by sending a message to an executor.

[1] Another possible abstraction level would be that implementation of a message transmission—whether it is executed as a procedure call like sequential languages[9], or as a enqueue operation to the target object—is controlled by a scheduler. However, abstraction level is so low that it makes the architecture implementation specific, although it would improve performance in some cases.

```
[class metaobj (class args
                &key (executor some default))
 (state [mqueue := (make-queue)]
        [class := class]
        [state := ... ]
        [executor := [new executor]]
        [mode := ':dormant])
 (script
  ;; arrival of a message
  (=> [:message M R S]
      (queue-put (make-message M R S) mqueue)
      (when (eq mode :dormant)
        (accept-message)))
  ;; end of a script execution
  (=> [:finished]
      (if (queue-empty? mqueue)
        (setq mode :dormant)
        (accept-message)))
  ;; other meta-level operations are omitted
  )
 (routine ;; acceptance of a message
  (accept-message ()
   (let* ((m (queue-get mqueue))
          (c (find-script m class state))
          (thunk (make-thunk c Me)))
     (setq mode :active)
     [scheduler <=  [:req-exec thunk]])))]
```

Figure 6: Definition of Default Metaobject

2.3.3 Metaobject

Metaobject is a meta-level object that represents structural and object-specific aspects of an associated base-level object. It is designed so that the user can do customizations/introspections like follows: (1) the message queue can be inspected—this facility is useful to obtain the load value of individual objects; (2) the state variables can be inspected—useful to obtain application level information from the meta-level; and (3) behaviors to the events on the base-level object (i.e., behaviors to a message reception, invocation of a script execution, and end of script execution) can be customized—useful to attach information into an executable script and to record information, which is used for DRM systems.

Based on this observation, the definition of metaobject is in an event driven style(Figure 6). Basically, this definition is similar to the one in ABCL/R[12] and ABCL/R2[6]. The major difference is that our definition has a reference to a class object to hold the script data, while metaobjects directly hold the script data in ABCL/R and /R2.

2.4 Interface to Run-time Systems

The meta-level objects, such as the scheduler, are abstractions of the *modifiable* components of the run-time system. Not only the modifiable, but also interface to non-modifiable components is defined in our architecture. Those include primitive operations for object migration, information on the run-time system such as number of remote messages issued, information on the hardware configuration such as network topology. Note that the implementations of meta-level components that can be accessed by the above interfaces are not open. Rather they are closed so that low-level optimizations can be applied as much as possible.

3 Preliminary Evaluation

A prototype system is constructed on a multicomputer AP1000, which is a distributed memory parallel computer consisting of 32–1024 sparc processors running at 25MHz. The system translates an ABCL/R3 program to C functions. Its reflective features are limited from the one discussed in the previous section: by now, only the ability to control the object creation node and the mechanism to use user-defined scheduler is available. No compiler that collapses customized meta-level into efficient single-level code is investigated yet.

Using this prototype system, we carry out several experiments to measure (1) the cost for changing the scheduler, and (2) the effect of user-defined object allocation policy for tree computations. We have also evaluated the reflective architecture through describing various DRM systems, such as dynamic load balancing, which appear in [4].

3.1 Overhead to Change Scheduler

To evaluate the overhead to change schedulers, elapsed times for a null method invocation and a context switch are measured with different schedulers. In this evaluation, the program is executed on a single node. Used schedulers are: (1) the default scheduler using C function calls for a node-local message sending (as in *StackThreads*

implementation[10]), (2) the default scheduler using an active object queue, (3) the user defined scheduler using a queue. Note that the user defined scheduler is written in ABCL/R3 itself, and programmers need not write C functions.

Scheduler	time/call (μsec.)
(1) default, function	47.8
(2) default, queue	104.0
(3) user defined	190.0

Table 1: Elapsed Time for Null-Method Invocation with Different Schedulers

Table 1 shows elapsed time for a method invocation with different schedulers. With (1), it is as twice faster than as the one with (2). This is because, messages are passed as C-function arguments in the former case, while the latter case pays the cost for message allocation on the heap memory, and queue manipulations. With the user defined scheduler, a method invocation takes additional 90μseconds to (2). This is because, two more method calls to the scheduler (when an object becomes active and when the next active object is scheduled) are performed in addition to (2)'s case.

The benchmark shows that the overhead due to the user defined scheduler is not small enough. However, there are plenty of room for optimization, because the current implementation is very slow in the basic operations. A node local (null) method invocation takes about 50μseconds in our implementation, while a few μseconds in the highly optimized implementations like [10]. Therefore, when we use more efficient implementation, such overhead could be kept fairly low.

Furthermore, we are convinced that most of overheads can be eliminated by collapsing the meta-levels using partial evaluation techniques. It makes easier to reasoning about the modifications to the meta-level that the meta-level is defined in an operational way.

3.2 Locality Control

The locality control is a simple DRM example for tree style computations. The basic idea is that for an object at the shallow level in a search tree, its child objects are created at randomly-chosen nodes; while for objects at the deep level in the tree all its child objects are created at the same node to the creator's.

The target application program used here is the N-Queens problem in which a node in a search tree is implemented as a concurrent object of ABCL/R3. Since depth in the search tree is not explicitly available in the base-level program, it is maintained by metaobjects. A metaobject is created with a depth parameter in addition to the other default parameters. The metaobject sends the depth parameter whenever it requests an execution to a scheduler. The parameter is passed along with an ordinary context of a script execution, such as environment and metaobject's ID. On an object creation, the object executor determines a node where new object will be created, based on the depth parameter and a pre-determined threshold.

Table 2 shows the elapsed times for the execution with the different level of depths on 64-nodes AP1000. The first row in each table shows a result *without* the locality control; all object creations are remote. With locality control system, we observe about two-fold speed-ups in the best cases.

4 Conclusion

In this paper, we have proposed an object-oriented concurrent reflective language ABCL/R3. Unlike its predecessor, notion of nodes, which corresponds to distributed memory processor elements of multicomputers, and the scheduler, which governs the execution order in each node, are explicitly available at the meta-level. These abstractions make it possible to describe various DRM systems for irregular concurrent applications, in an encapsulated and portable way. This paper also presents preliminary evaluation using a prototype implementation of the language running on a multicomputer. The benchmark results show that the overhead from using a user defined scheduler is kept as low as two or three null method calls for each method invocation, and that a simple locality control system ex-

12-Queens

threshold depth	elapsed time (sec.)	utilization(%)
—	13.88	97
8	7.655	95
7	6.595	93
6	6.541	87
5	7.301	77

11-Queens

threshold depth	elapsed time (sec.)	utilization(%)
—	2.742	95
7	1.511	92
6	1.403	88
5	1.524	81
4	1.917	70

Table 2: Benchmark Results for Different Threshold Depth Parameters

hibits about two-fold speedup compared to the simple one.

References

[1] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of ECOOP*, 1993.

[2] G. Kiczales, J. Lamping, and A. Mendhekar. What a metaobject protocol based compiler can do for Lisp. Draft paper, 1994.

[3] D. B. Loveman. High performance Fortran. *IEEE Parallel & Distributed Technology*, 1(1):25–42, Feb. 1993.

[4] H. Masuhara. Study on a reflective architecture to provide efficient dynamic resource management for highly-parallel object-oriented applications. Master's thesis, Department of Information Science, Faculty of Science, University of Tokyo, 1994.

[5] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of OOPSLA*, pp. 127–145, Oct. 1992.

[6] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP*, 1991.

[7] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: Distributed programming system with multi-model reflection framework. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pp. 36–47, Nov. 1992.

[8] B. C. Smith. Reflection and semantics in Lisp. In *Conference record of POPL*, pp. 23–35, 1984.

[9] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of PPOPP*, May 1993.

[10] K. Taura, S. Matsuoka, and A. Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs. In *JSPP*, pp. 25–32, May 1994.

[11] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[12] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA*, pp. 306–315, Sept. 1988. (revised version in [13]).

[13] A. Yonezawa ed. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.