An Object-Oriented Approach to Software Development for Parallel Processing Systems

Stephen S. Yau, Xiaoping Jia, Doo-Hwan Bae, Madhan Chidambaram, and Gilho Oh
Computer and Information Sciences Department
University of Florida
Gainesville, Florida 32611-2024, USA

Abstract

Despite the rapid advances in development of parallel processing systems in recent years, the progress in software development to effectively utilize the parallel processing systems has been relatively slow. In this paper, an object-oriented approach to software development for parallel processing systems is presented. This approach is based on a computation model which incorporates the functional paradigm in the object-oriented paradigm and allows exploitation of massive parallelism without sacrificing the effectiveness of the object-oriented paradigm. The issues related to the development processes are discussed.

1 Introduction

In recent years, a variety of powerful parallel processors have been introduced. However, the progress in software development to effectively utilize the parallel processing systems has been relatively slow. Among the many approaches to software development for a parallel and distributed system, including dataflow-oriented, communication-oriented and object-oriented approaches, the object-oriented approach is considered a more promising approach [1]. In the object-oriented approach [2, 3], a software system is modeled as a set of cooperating objects that can communicate with each other through a unified communication mechanism. Each object corresponds to an entity such as data, action or hardware device. This paradigm can naturally reflect the structure of the problem space, and thus is suitable for representing inherently concurrent behavior. In addition, the data abstraction and inheritance mechanism in the object-oriented paradigm enhance the comprehensibility, extensibility, maintainability, modularity and reusability of the software systems. However, the traditional object-oriented paradigm suffers from two deficiencies that severely limit the parallelism: 1) The execution of a program is initiated by activating only a small number of objects and 2) the sender of a message becomes inactive until it receives a response from the receiver of that message. Although some attempts have been made to enhance the parallelism in this paradigm [4-7], massive parallelism is still not obtainable in this paradigm.

On the other hand, the purely functional paradigm offers great potential for parallel execution [8]. Functional languages are well known for their brevity and elegance. Parallelism is implicit in functional programs due to the *referential transparency*. Thus the programmers are liberated from the complications caused by the parallelism and the software development effort can be reduced. Functional programs are more amenable to formal verification than imperative programs. However, the purely functional paradigm is history insensitive and hence has limited expressive power.

In this paper we will present an approach to software development for parallel processing systems based on a parallel object-oriented computational model PROOF [9], which incorporates the functional paradigm in the object-oriented paradigm, and allows exploitation of parallelism at various levels of granularity without sacrificing the effectiveness of the object-oriented paradigm. The objective of our approach is to reduce the software development effort for parallel processing systems and explore the parallelism in the software systems. These will be achieved by supporting information hiding, modularity, modifiability and reusability and allowing exploitation of parallelism at various levels of granularity. Our design approach will be presented and the issues on parallelization, allocation and optimization will be discussed. An example will be given to illustrate our approach.

2 Our Design Approach

Our design approach integrates the object-oriented and functional paradigms at the following two levels:

- Object level — PROOF supports all the important features in the object-oriented paradigm, including objects, classes, and inheritance.
- Method level — Methods of objects are defined as applicative functions. Due to the referential transparency of applicative functions, fine grain parallelism can be exploited.

In PROOF, an object class is defined by a pair of class interface and class definition. The class interface provides only the necessary information for invoking

0730/3157/91/0000/0453/$01.00 © 1991 IEEE

its methods. The implementation of the class is encapsulated in the class definition. Subclasses can be defined by inheriting methods from their superclasses. In PROOF, a software system is represented as a set of concurrent objects, which are instances of classes. Two different types of objects are distinguished.

- Active object - an active object is active by itself and may invoke the methods of other objects, i.e. activates other objects.
- Passive object - a passive object is active only when its methods are invoked by other objects.

Each active object is associated with a *body*. The concept of bodies is similar to the concept of sequential processes. Each body is a function. The evaluations of the bodies are activated when the system is initiated. Objects are persistent. Modifications to objects are achieved by the *reception* construct, which makes the computation model history sensitive and preserves the referential transparency at the method level. Synchronization among concurrent objects is achieved by the *guards* attached to the methods. The guards are defined in such a way that they are directly inheritable with the methods with which they are associated. Simultaneous accesses to the same object are permissible, but simultaneous modifications to the same object must be serialized; otherwise, it may result in an inconsistent or incorrect state of the object. The serializability can be guaranteed by the locking mechanism adopting the two phase locking protocol [10]. For each expression involving objects, proper locks must be obtained for each object prior to the evaluation of the expression. The locking mechanism will be discussed in the following section. These issues are discussed in detail in [9]. Our software development approach is based on PROOF and how the features of PROOF are used in our approach will be discussed in detail in the following sections:

The Design Process

The object-oriented design process consists of the following steps:

Step 1 Identify objects and classes.

Step 2 Determine class interfaces.

Step 3 Identify dependency and communication relationships among objects.

Step 4 Determine class composition and methods.

Step 5 Determine active object bodies.

In Step 1, objects in the problem domain are identified by analyzing the semantic contents of the requirement specification of the system to be developed. Usually, each object corresponds to a real-world entity, such as sensors, control devices, data, and actions. The software system is represented by a set of communicating objects. The active and passive objects are distinguished.

In Step 2, object class interfaces are determined. In PROOF, every object is considered as an instance of an object class. Therefore, instead of defining objects directly, the object classes to which they belong must be defined. The interface of an object class consists of the specifications of the methods provided by the class. For each method, its specification consists of the input and output parameters and their types. The actual definitions of the methods are hidden and will be defined in a later stage. The class interface definition in PROOF is slightly different from that in the conventional object-oriented approach. Let m be a method of class C. In conventional object-oriented approach, the specification of m may appear as follows:

$$m : I \mapsto O$$

where I is the input parameter(s) of m and O the output parameter(s) of m. Typically, m will also have side-effects on the internal states S of the instances of C. In PROOF, the methods are defined as applicative functions. Therefore, no side-effects are allowed. The internal state of an object will be an explicit input and/or output parameter of m if the internal state is accessed and/or modified. Typically, in PROOF the interface of m will appear as follows:

$$m : I \times S \mapsto O \times S$$

The methods will not directly modify the state of the objects. Instead, a new state of an object will be returned when the object needs to be modified. The modification of objects will be achieved by a special construct discussed below.

In Step 3, the dependency and communication relationships among the objects are identified based on the method invocations among objects. The dependency and communication relationships are described using the object communication diagram (see the example below).

In Step 4, the composition and the methods for each object class are determined. The class definition consists of the composition part and the method part. The composition part defines the internal data structure of the class. Various constructors, such as list and Cartesian product, are provided. A typical functional style is adopted in the method definition. A rich set of functional forms, i.e. high-order functions, as well as primitive functions are predefined. The definition of a method of an object consists of a *guard* and an *expression*. The guard, which is a predicate, specifies the synchronization constraint and the expression statement specifies the behavior of the method. The guard attached to a method is defined in a way that it only depends on the status of the local data, but does not depend on the definition of other methods. Therefore, the guards are directly inheritable with the methods. The expression is a purely applicative function. Due to the referential transparency of applicative functions, fine grain parallelism can be exploited.

In Step 5, active object bodies are determined. A body will be associated with each active object. The body of an active object is in the form $f_1//f_2//\ldots//f_k$, where each f_i is a function and all k functions separated by // are evaluated simultaneously. // is a parallel construct indicating parallel execution. f_i can be recursively defined and can be diverse. Thus, the evaluation process may be infinite. In f_i, methods of objects can be invoked, and the states of objects may be modified. The modification of an object is achieved by the reception construct, in the form $R[|o|]e$, where o is an object name and e is an expression with applications of purely applicative functions only. The reception construct can only occur in the bodies of active objects. It means that the object o will receive the value of the expression e. This construct modifies the states of the object. It differs from the conventional assignment in the following aspects:

1. The expression e contains only purely applicative functions. Thus, the evaluation of the expression e is side-effect free and can be parallelized.
2. The expression e must return a new state of the object o. o receives the new state as a whole entity. Therefore, no partial modification and no inconsistent state of the object is possible.

The object o may be composed of other objects. However, the composing objects cannot appear in the reception construct as a recipient. The overhead involved in complying with this no partial modification rule can be minimized by optimization based on static data dependency analysis.

An Example

Our approach is illustrated by the example warehouse management system. The following is a brief statement of the requirements of the warehouse management system:

> The warehouse management system interacts with manufacturers and customers such as retailers, and manages and controls the warehouse. Manufacturers generate items (goods) and send them to the warehouse manager and items are stored on warehouse-racks. The warehouse manager retrieves items from warehouse-racks and sends them to the customers upon their requests. The capacity of this warehouse is fixed. Reports of transaction information are generated periodically. The number of manufacturers, customers and warehouse-racks may vary from time to time.

Step 1: According to the above statement, the following object classes are identified in the warehouse management system: `Rack`, `Transaction_list`, `Transaction_item`, `Producer_window`, `Consumer_window`, `Manager_class`, `Producer_class`, `Consumer_class` and `Reporter_class`. Among them, `Manager_class`, `Producer_class`, `Consumer_class` and `Reporter_class` are active object classes, and the remaining object classes are passive.

Step 2: As an example, the interfaces of `Rack` in the warehouse management system is shown in Fig. 1. The method `put` has `Rack` and `itemtype` as its input parameters and `Rack` as its output parameter. The method `get` has `Rack` as its input parameter and `Rack` and `itemtype` as its output parameters. The interfaces for all the object classes identified in Step 1 need to be defined.

Step 3: The dependecy and communication relationships in the warehouse management system are identified and described using an object communication diagram. In the object communication diagram, active objects are represented as rounded rectangulars, and passive objects are represented as rectangulars. The links between the objects indicate the communication, i.e. method invocation, between objects. The arrows on the links indicate the direction of invocation. Fig. 2 is the object communication diagram for the warehouse management system. The `producer_window` and `consumer_window` are designed for the interfaces between the `warehouse_manager` and the `producer` and between the `warehouse_manager` and the `consumer` respectively. For the sake of illustration, the design is simplified and only one instance of each of the following object calsses, `Producer_Class`, `Producer_Window`, `Consumer_Window`, `Consumer`, and `Rack`, is shown in the diagram.

Step 4: As an example, the definition of class `Rack` in the warehouse management system is shown in Fig. 3. The composition clause defines the local data of the class `Rack`, which consists of a list of items and a counter of the unmber of items in the rack. `inc`, `dec`, `append_right`, `tail`, and `head` are all predefined functions. `inc` and `dec` increments and decrements the input parameter respectively. `append_right` appends its first parameter to the right end of its second parameter, which must be a list. `tail` has a list as its input parameter and returns a new list which is the same as the input list except the first element is removed. `head` has a list as its input parameter and returns the first element of the input list. `Beta` is a predefined functional form, which has a list of functions and a list of arguments as input parameters, and returns a list of values obtained by applying each of the functions in the first list to each of the argument in the second list.

```
class  Rack;
  method put :: Rack -> itemtype -> Rack;
  method get :: Rack -> Rack X itemtype;
end class
```

Figure 1: The interface of class `Rack`.

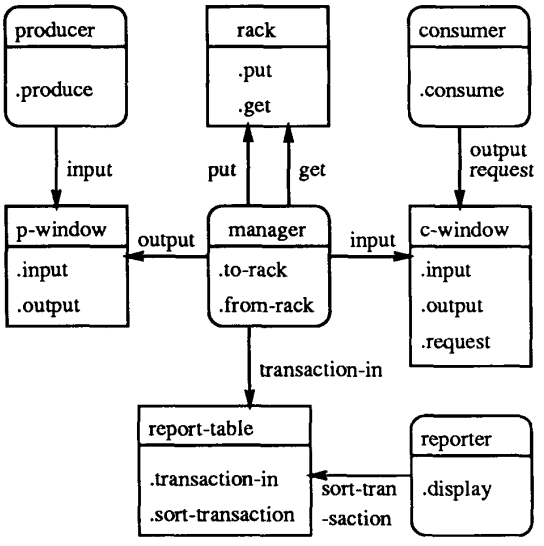

Figure 2: The object communication diagram for the warehouse management system.

```
class definition  Rack(size,itemtype)
      composition
           r:list(itemtype) X count:int
      method put r x
             guard(count < size)
             expression
             Beta[(append_right x),inc] r

      method get r
             guard(count > 0)
             expression
             [Beta[tail,dec] r, head(r.buf)]
      end class
```

Figure 3: The definition of class `Rack`.

Step 5: The body of the active object `Producer` in the warehouse management system is shown in Fig. 4. The active object `Producer` invokes its method `produce` and the passive object `p_window`'s method `input`. After invoking those methods, it modifies the passive object `p_window`. The body is defined using a recursive function `while`, which is diverse in this case. Thus, the body for `Producer` is a non-terminating process and will be activated when the system is first initiated.

We have illustrated the software design approach based on PROOF. The design can be translated to the implementation of the system in some target language. The transformation process will be discussed in the following section.

```
Active Object Producer: instance of
                        Producer_class
 Body  while(True,R [| p_window |]
            (input(p_window, produce)))
```

Figure 4: The body of active object `Producer`.

3 Transformation

The transformation involves the following three major aspects: parallelization, allocation, and optimization. We will discuss each of these three aspects.

Parallelization

The goal of parallelization is to make the implicit parallelism in PROOF model explicit. Software designs based on PROOF will be translated into equivalent forms in some intermediate language with explicit parallel constructs. The parallelization process consists of the following two phases: (a) generate a lock manager for each object, and (b) generate parallel code for the body of each active object.

Lock managers

A lock manager is associated with each passive object and each active object with local data. Each lock manager is an independent parallel unit. Any object that wishes to access another object by invoking one of its methods must first issue a request to the associated lock manager for an appropriate lock. There are three types of locks: **R-Lock** is requested when the expression only needs to read the value of the object. **W-Lock** is requested when the expression is expected to modify the state of the object in the near future. **M-Lock** is requested just before the modification of the state of the object is made. The compatibility of the locks is shown in Table I.

Table I Compatibility of Locks

	R-Lock	W-Lock	M-Lock
R-Lock	C	C	NC
W-Lock	C	NC	NC
M-Lock	NC	NC	NC

C: Compatible, NC: Not Compatible

The lock manager will either approve or deny a lock request based on the compatibility chart in Table I. A lock request to an object is approved only when the previously approved locks to the same object are compatible to the current request. When the lock request is approved, the method invocation may proceed; otherwise the method invocation must wait until the lock request can be approved.

Bodies of active objects

```
parbegin
      process(f1);
  ||  process(f2);
      :
  ||  process(fk);
parend
```

Figure 5: The translation template for active object bodies.

```
Request W-Lock for object o;
Request R-Lock for objects associated with e;
para_eval(v, e);
Release lock for objects associated with e;
Upgrade to M-Lock for object o;
o := v;
Release lock for object o
```

Figure 6: The translation template for the reception construct.

The bodies of active objects are in the form: $f_1//f_2//\ldots//f_k$. Each f_i is a function which may contain the reception construct R. Each f_i will be translated into a process as shown in Fig. 5. *process*(f) translates a function f into its implementation as a sequential process. This translation is similar to the translation of functions in compiler for sequential functional languages.

The translation template for the reception construct $R[|o|]e$ is shown in Fig. 6. Because e is an expression with only applications of applicative functions, the evaluation of e can be parallelized. v is a fresh variable. *para_eval* will generate code for *parallel* evaluation of e and the result will be stored in the variable v.

The basic schema for parallelizing expressions with only applications of applicative functions is shown in Fig. 7. The subexpressions can be evaluated in parallel with the evaluation of the main expression. When the value of one of the subexpressions is needed by the main expression, the evaluation of the main expression must wait until the value of the subexpression is available, i.e. the evaluation of the subexpression is completed. When the evaluation of the main expression is completed, it will kill all the evaluation processes for its unfinished subexpressions. This will achieve the same effect as lazy evaluation. Therefore, the non-strict semantics can be supported. The *para_eval* transformation can be applied recursively to each of the subexpressions.

Allocation

A program can be composed of a set of tasks having dependency relationships. Allocation is to distribute such tasks to the processors of a multi-processor system. The goal is to minimize the execution time of the program. The major factors to be considered in the allocation process are the number of processors available, the number of communication links connected to each processor, the topology of the system, the execution time of each parallel components, and the communication cost.

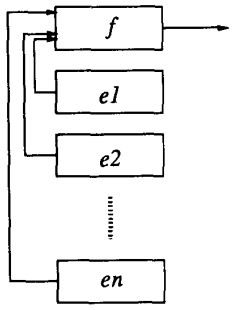

Figure 7: Parallelization schema for applications of applicative function $f(e_1, e_2, \ldots, e_n)$.

The allocation can be done either statically or dynamically. Static allocation is done at the translation time. Since the exact behavior of the program is unknown at the translation time, the allocation can only be based on estimation of the execution time and the communication cost. It is very difficult for static allocation to achieve the optimal performance of the program. Static allocation is generally based on heuristics. Dynamic allocation will adjust the load of processors at run-time. It has the advantage of knowing the exact behavior of the program. However, a run-time manager is needed, and run-time overhead is also involved.

One of the issues involved in allocation is the determination of the grain size of parallelism. Large grain size leads to limited parallelism and low ratio of utilization of the processors available. Small grain size causes excessive inter-processor communication and hence increases the communication cost. The determination of grain size will be one of our major research topics in this area.

Currently, we adopt a simple clustering allocation strategy for PROOF. The strategy is based on the object communication diagram. When the number of objects is smaller than the number of processors, we partition the processors into a set of groups and assign each group to an object to exploit fine grain parallelism. When the number of objects are greater than the number of processors, we cluster the objects having frequent communication into one and assign one cluster of objects to each processor. Although this strategy is very primitive, it is easy to implement. However, even if the number of the objects is greater than the number of nodes, the number of objects executing simultaneously at any moment is typically smaller than the number of the objects in the whole system. Hence, it is necessary to exploit

fine grain parallelism so as to fully utilize the power of parallel processing systems. More effective allocation strategies that allow exploitation of both coarse and fine grain parallelism are being studied.

Optimization

At source code level, the following typical optimization techniques will be applied: common subexpression extraction, useless expression removal, tail recursion removal, and function call in-lining, etc. During the parallelizing process, data dependency analysis will be performed to eliminate unnecessary data movement. In particular, to eliminate the overhead due to prohibition of partial modification, static data dependency analysis will be performed and only a minimum amount of data necessary for modification will be computed and moved so that communication and updating costs can be minimized. During the allocation process, according to the topology of the underlying hardware, various kinds of code movement will be performed to reduce the amount of data to be transmitted among the processors, and thus the communication cost is reduced.

4 Discussions

We have presented an object-oriented approach to software development for parallel processing systems based on an integrated object-oriented and functional computation model – PROOF. This approach has been illustrated using the warehouse management system. The implicit parallelism in PROOF is made explicit through the transformation process. This approach can fully exploit the parallelism in software systems and also reduce the development effort by supporting software engineering principles such as information hiding, modularity, modifiability and reusability.

A prototype design language based on PROOF, called PROOF/L, has been developed. The translator from PROOF/L to OCCAM is currently under development. The implementations generated from designs in PROOF/L will run on a 16-node transputer system. We are investigating the proper intermediate representation for PROOF/L, which will enable us to generate implementations on a variety of different architectures and target languages. The locking mechanism adopted currently is very strict. Consequently, it could degrade performance, and hence a more sophisticated locking mechanism is desirable. Allocation strategy significantly affects the performance of parallel processing and will be a major topic for future research. In particular, we need to determine the proper grain size of parallelism so that we can exploit considerable parallelism without degrading the performance due to communication cost. In addition, various software tools such as program execution monitor and debugger need to be developed to support our approach.

References

[1] Yau, S. S., Jia, X., and Bae, D.-H., "Trends in Software Design for Distributed Computing Systems," *Proc. Second IEEE Workshop on Future Trends of Distributed Computing Systems,* October 1990, pp. 154-160.

[2] Booch, G., "Object-Oriented Development," *IEEE Trans. on Software Engineering,* Vol. SE-12, No. 2, February. 1986, pp. 211-221.

[3] Jackson, I., "Object-Oriented Development in an Industrial Environment," *Proc. Object-Oriented Programming Systems, Languages and Applications,* ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 183-191.

[4] America, P., "A Parallel Object-Oriented Language," *Object-Oriented Concurrent Programming.* In Yonezawa, A and Tokoro, M. (Eds.). MIT Press, Cambridge, MA, 1987, pp. 199–220.

[5] Yokote, Y., and Tokoro, M., "Concurrent Programming in Concurrent Smalltalk," In Yonezawa, A. and Tokoro, M. (Eds.). *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, MA, 1987, pp. 129–158.

[6] Kafura, D. G., and Lee, K. H., "Inheritance in Actor Based Concurrent Object-Oriented Languages," *Proc. Third European Conference on Object-Oriented Programming.* 1989, pp. 131–145.

[7] Tomlinson, C., and Singh, V., "Inheritance and Synchronization with Enabled-Sets," *Proc. Object-Oriented Programming Systems, Languages and Applications.* October. 1989, pp. 103–112.

[8] Hudak, P., "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys,* Vol. 21, No. 3, September 1989, pp. 359–411.

[9] Yau, S. S., Jia, X, and Bae, D.-H., "PROOF: A Parallel Object-Oriented Functional Computation Model," *Journal of Parallel and Distributed Computing,* 1991, *in press.*

[10] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database System," *Communications ACM.* Vol. 19, No. 11, November 1976, pp. 624–633.