An Architecture-Independent Software Development Approach for Parallel Processing Systems*

Stephen S. Yau, Doo-Hwan Bae† and Jun Wang

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406, USA

Information and Communication Engineering Department
Korea Advanced Institute of Science and Technology
Seoul, Korea

Abstract
In this paper, an architecture-independent software development approach for parallel processing systems is presented. This approach is based on the parallel object-oriented and functional computation model PROOF and separates the architecture dependent issues from software development. It also facilitates software development for any parallel processing systems by relieving the programmers from the consideration of processor topology and various parallelization aspects of the software. Overall, this approach allows the exploitation of parallelism at both levels of granularity: object level and method level, thereby making our approach effective for software development for various MIMD computers. Software developed using our approach will reflect the parallel structure of the problem space which will make the software more understandable and modifiable. A framework consisting of object-oriented analysis, object-design, coding and transformation phases is presented for software development for parallel processing systems. An example is given to illustrate this approach.

keywords: architecture-independent software development, object-oriented paradigm, functional paradigm, parallel processing.

1 Introduction
In spite of vast increase in speed of various parallel processing systems, the potential for high performance computing systems for various applications such as command, control, communication, and intelligence systems, space exploration, weather prediction, and telecommunication systems, where there are many interacting components, shared resources, and computationally intensive tasks, cannot be realized without effective software development methods for such systems. Unfortunately such methods are far from being mature due to the complexity of concurrency and synchronization issues. The lack of such methods is a major obstacle for the effective use of parallel processing systems in various application areas. The existing software development approaches [1, 2, 3, 4] for parallel processing systems focus on the exploitation of data parallelism in numerical computation and do not address design steps for developing parallel software. The well-known existing OOA/OOD methods, such as OMT [5], Booch [6], etc do not address software development issues for parallel processing systems. Therefore, they are not suitable to solve more general applications.

In this paper, we will present an architecture-independent approach for software development for parallel processing systems. It is based on the parallel object-oriented functional computation model (PROOF) [7] which incorporates the functional paradigm in the object-oriented paradigm. The object-oriented paradigm reflects the parallel object structure for the problem space and is suitable for representing inherently concurrent behavior, which makes the software more understandable and modifiable. The functional paradigm facilitates us to exploit method level parallelism. The main advantage of this approach is that this methodology separates the architecture dependent issues from software development. Hence, the programmer does not need to be concerned with the issues such as synchronization, parallelization or the network topology of parallel processing systems thereby making the software development independent of the architectures of parallel processing systems [8]. This approach will allow the exploitation of parallelism at both levels of granularity, coarse-grain at object level and fine-grain at method level, and is suitable for MIMD machines. An example will be given to illustrate our approach.

2 Overall Framework
Our approach to software development for parallel processing system is based on the computation model PROOF which incorporates the functional paradigm into the object-oriented paradigm. In PROOF, each object is an instance of a class, and can be either passive, active or pseudo-active. A passive object acts like a service agent. It waits passively until one of its methods is invoked by some other objects. An active object is active initially, and it may remain active throughout its execution, except for occasional suspensions for synchronization with other objects. A pseudo-active object is invoked by other objects, and in turn invokes methods of other objects. A body will be attached to each active and pseudo-active objects. Bodies of objects are functions whose roles are to invoke the methods and modify the states of the objects. A class is defined by its interface and definition. The class interface describes the signatures of the methods provided by the class. The class definition consists of the local data and the

0730-3157/95 $04.00 © 1995 IEEE

370
implementation of methods of the class. The synchronization among the objects is achieved by attaching an optional precondition guard to the methods in a class. Each guard is a predicate. The object which invokes the method is suspended when the attached guard is evaluated to be False, and it is resumed when the guard predicate becomes True. The guard attached to a method is defined in a way that it only depends on the status of the local data, and does not depend on invocations of any other methods. Methods are defined as purely applicative functions or functional forms, i.e., high-order functions. Thus, it is easy to detect and exploit massive parallelism due to referential transparency.

Our framework consists of the following phases: object-oriented analysis, object design, coding in PROOF/L, and transforming PROOF/L code to a target language of a parallel processing system. It is shown in Figure 1.

In the object-oriented analysis phase, the system is represented by a set of objects and the interrelationship among the objects.

In the object design phase, the objects identified in the object-oriented analysis phase are designed using the notations defined in PROOF. The class interface definitions and information about the object behavior are used to design the objects. The design of the objects has to be analyzed and verified for various liveness and safeness properties. For this purpose, we transform our design into Petri nets, which were used in our approach mainly because our design can be easily represented in the Petri-net model and many techniques are available to analyze Petri-net models.

In the coding phase, the design of the software system is implemented in PROOF/L. The coding in PROOF/L is straightforward and will not be discussed in this paper.

In the transformation phase, PROOF/L code is translated to a target code to be executed on a parallel processing system. The transformation of the program in PROOF/L into a target language involves the following major steps as shown in Figure 1: partitioning, front-end translation, grain size analysis, back-end translation, and allocation. These steps can be divided into two groups, one for front-end translation and partitioning dealing with architecture independent issues, the other for grain size determination, back-end translation, and allocation dealing with architecture dependent issues. In the partitioning step, the objects in the software system are partitioned into a set of clusters [9]. The objective of our partitioning approach is to improve the performance of the software by reducing communication cost among processors while maintaining the parallelization among objects. During this phase, we are only concerned with the object level parallelism. In the front-end translation, PROOF/L code is translated into IF1 (Intermediate Form 1), a task precedence graph. Then, grain size analysis is performed to determine the proper size of tasks to be executed on different processors in order to improve the performance of execution, and a modified IF1 code is generated. In the back-end translation, the modified IF1 code is translated into the target code. Architecture-dependent issues, such as communication overhead and processor load, are considered in the back-end translation. Then, the target code is allocated to processors. The details of the grain size analysis and transformation are given in [11].

3 Object-Oriented Analysis

Our approach starts from the problem statements given in a natural language. We assume that the problem statements are complete in the functionalities of the problem without ambiguities. The approach given in [12] may be used to clarify the problem statements.

The object-oriented analysis phase consists of the following steps.

1) Identify objects and classes.
2) Determine class interfaces.
3) Specify dependency and communication relationships among objects.
4) Identify active, passive and pseudo-active objects.
5) Identify the shared objects.
6) Specify the behavior of each of the objects.
7) Identify bottleneck objects, if any.
8) Check the completeness and consistency.
In Step 1), objects are identified by analyzing the semantic contents of the requirement specifications. All physical and logical entities are recognized as object candidates. Each object corresponds to a real-world entity, such as sensors, control devices, data and actions. Objects having similar behavior can be grouped together to form classes. The nouns in the specification can be the candidates for the objects and the verbs for the operations \[13\]. The candidates for objects can also be detected from the data flow diagram\[14\].

In Step 2), object class interfaces are determined. Because every object is considered as an instance of an object class, instead of defining objects directly, the object class to which they belong must be defined. The interface of an object class consists of the signatures of the methods provided by the class. For each method, its signature consists of the input and output parameters and their types.

In Step 3), the relationships among objects are specified using the object communication diagram, in which the objects are represented as rectangles, the links between the objects indicate the communication between objects, i.e., method invocations, the arrows on the links indicate the directions of invocations, the methods defined in an object as interface are written within the object with the method names beginning with a period, and the labels on the arrows show which methods are being invoked by an object. An example of the object communication diagram is shown in Figure 3.

In Step 4), the objects are classified to active, passive or pseudo-active objects according to their invocation properties. They can be easily identified from the object-communication diagram. The active objects have outgoing arrows with optional incoming arrows for responses from other objects, the passive objects have only incoming arrows, and the pseudo-active have both incoming and outgoing arrows.

In Step 5), the shared objects are identified. An object is a shared object if it has the local data which can be accessed by a number of objects. There are two kinds of shared objects: read-only and writable. A read-only shared object has the local data which cannot be modified by other objects. A writable shared object has the local data which can be modified by other objects. Read-only objects can be freely duplicated as many times as desired. However, writable objects should not be duplicated easily because maintaining the consistency of the data will cause additional overhead. Shared writable objects may be bottleneck objects as they may have to be executed sequentially to maintain the consistency of the states of the objects.

In Step 6), to specify the behavior of an object, we use the notations similar to those in \[15\]:

- \( \text{SEQ}(m_1, m_2, \ldots, m_n) \): The methods \( m_1, m_2, \ldots, m_n \) are executed sequentially in the order \( m_1, m_2, \ldots, m_n \).
- \( \text{CON}(m_1, m_2, \ldots, m_n) \): The methods \( m_1, m_2, \ldots, m_n \) are executed concurrently.
- \( \text{WAIT}(m, O) \): An object is waiting for the invocation of its method \( m \) by another object \( O \) to proceed with its execution.
- \( \text{SEL}(C; m_1, \ldots, m_n) \): An object selects one of the methods for execution from the methods \( m_1, m_2, \ldots, m_n \) based on a condition \( C \). It behaves similar to the CASE statement in ordinary programming languages.
- \( \text{ONE-OF}(\text{WAIT}(m_1, O_1), \ldots, \text{WAIT}(m_n, O_n)) \): An object permits only one of its methods to be invoked by other objects among the set of methods \( m_1, m_2, \ldots, m_n \) defined in the object. It is typically used to describe the behavior of shared writable objects, which can serialize method invocations from other objects that arrive simultaneously. \( \text{ONE-OF} \) will always be associated with a WAIT construct in a shared writable object because the object \( O \) will have to wait for other objects to invoke its methods.

In Step 7), the bottleneck objects, which may degrade the performance of the software system to be developed, are usually shared writable objects and identified from the description of the object behavior in the above step. If such an object is found, then redo or refine the object-oriented analysis to reduce the bottleneck if possible to prevent these objects from limiting parallelism. This step may increase the number of objects in the software system. Repeat Steps 2) to 6) until the object-oriented analysis is found satisfactory.

In Step 8), the result of the object-oriented analysis is verified with the given user requirements from which the possible threads of controls are identified, and each of them is examined using the behavior of the objects specified in Step 6). The verification can be done as follows \[16\]: First, the results of the object-oriented analysis are transformed into the specification using a formal specification language, which is converted into an information tree. Then, the completeness and consistency between the results of the object-oriented analysis and the user requirements can be verified by comparing the information tree with the user requirements.

### 4 Object Design

In our approach, the object design is specified using the notations defined in PROOF\[7\]. The class interface, definition, and information about the object behavior are used to design the objects. Our approach to object design involves three steps:

1) Establish the class hierarchy.
2) Design the class composition and the methods in each object.
3) Design the bodies of the active and pseudo-active objects.

In Step 1), a set of operations and/or attributes that are common to more than one class can be abstracted and implemented in a common class called the super-class. The subclasses then have only the specialized features. Many human decisions are involved in building the class hierarchy. However, the similar method/data patterns presented in different class interfaces can provide hints for the developers to effectively derive superclass/subclass relations. In some cases, a superclass...
can be extracted from a single subclass and put in the
class library if needed. Establishing a class hierarchy
in the form of superclasses and subclasses increases the
reusability in the application. Class hierarchy also en-
hances the modularity and the extensibility of the soft-
ware system.

In Step 2), the composition of the local data and the
methods for each object class are designed. The com-
position defines the internal data structure of the class.
Various constructors, such as list and Cartesian prod-
uct, are provided. A typical functional style is adopted
in the method definition. In the method design, the in-
ternal state of the object to which the method belongs
is included as both the input and output parameters so
that side-effects can be avoided. A method of an object
consists of an optional guard and an expression. The
synchronization among concurrent objects is achieved
by the guards attached to the methods. The guard at-
tached 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.
Due to the referential transparency of applicative func-
tions, fine-grain parallelism can be exploited.

Selection of algorithm and data structure is an im-
portant part of the method design. The selection of al-
gorithms to accomplish a specific task should be based
on certain criteria which satisfy the required constraints
such as accuracy, timing requirements, use of common
utilities across the design, reuse of previously developed
software, computational complexity, flexibility, ease of
implementation, understandability, etc.

In Step 3), a body is associated with each active
and pseudo-active object. There is no body associated
with a passive object as it does not invoke any meth-
ods. The role of a body is to invoke a method and to
modify the state of the objects represented by their lo-
cal data. The body in each object is expressed in the
form e1//e2//...//en where each ei is an expression
representing method invocations and expressions sepa-
rated by // are evaluated simultaneously. // is a parallel
construct indicating parallel execution. ei can be recur-
sively defined and can be diverse. Thus, the evaluation
process may be infinite. In ei, methods of objects can
be invoked, and the states of objects may be modified.
The modification of an object is expressed by the re-
ception construct which has the form R[:O]c, where O
called a recipient object is an object name and c is an
expression with applications of purely applicative func-
tions. The reception construct can occur only in the
bodies of active and pseudo-active objects. The recep-
tion construct indicates that the object O will receive
the value returned by evaluating the expression e.

5 Transformation

The transformation of PROOF/L code to a target
code involves two steps as shown in Figure 2: The first
step is called front-end transformation, which makes the
implicit parallelism present in PROOF/L code explicit.
This step is a semantics-oriented transformation, and
architecture dependent issues are not involved. The sec-
ond step is called the back-end transformation. This
step is performance-oriented, and architecture depen-
dent parameters, such as communication types, num-
ber of links and number of processors, are used to per-
form various analysis. We have developed the front-
end translator from PROOF/L to IF1 and two back-end
translators from IF1 to nCube C and KSR C, respec-

6 An Example

In this section, we use an example of a warehouse
management system to illustrate our approach. A sim-
plicated version of the requirements specification of the
warehouse management system is given as follows: The
warehouse management system interacts with manu-
facters and customers, and manages the warehouse.
Manufacturers generate goods upon request from the
warehouse manager, and send them to the warehouse
manager. The warehouse manager store the goods in the
warehouse racks, and retrieves them to the customers
upon customer's request. The capacity of this warehouse
is fixed. Reports of transaction information are gener-
ated periodically.

6.1 Object-Oriented Analysis

1) From the requirements specification, the following
objects are identified: manager, producer, consumer, re-
porter and rack.

2) As an example, the interface of rack is shown as fol-
ows:

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

The method put has rack and itemtype as its input pa-
rameters and rack as its output parameter. The method
get has rack as its input parameter and rack and item-
type as its output parameters.

3) Dependency and communication relationships among
objects are identified and described in an object com-
unication diagram shown in Figure 3 (a).

4) consumer is an active object, manager and producer
are pseudo-active objects, and rack and reporter are pas-
nive objects.

5) rack and reporter are shared objects. For example,
the methods defined in rack - put and get - update the
local data in rack, so rack is a writable shared object.

6) The behavior of each object is specified. For example,
rack has two methods put and get, which are invoked by
manager. The behavior of rack is specified as follows:
ONE-OF(WAIT(put,manager),WAIT(get,manager)).

7) In manager, two control threads can be found: one is
initiated by manager to receive items from producer and
the other by consumer to retrieve items. Thus, manager
can be split into two objects as shown in Figure 3 (b).

8) Control threads are verified by starting from the ac-
tive objects, such as consumer, and tracing the methods
invocations based on the object behavior.

6.2 Object Design

1) In this example, each object is an instance of a dif-
ferent class, except p-manager and c-manager. We can define a
class called manager-class having the common method
transaction-in, that can be inherited to its subclasses,
p-manager and c-manager.
The perspective version of pseudo code for the nCube parallel machine for the get method generated through the front-end and back-end transformation is shown as follows:

```c
void rack_get(rack *buf, rack *return_rack, itemtype *return_itemtype) {
    /* self-loop for passing the guard */
    while (buf->count <= 0);

    /* parallel dispatch the tasks */
    dispatch ( tail(buf->r) -> return_rack->r ) ;
    dispatch ( dec(buf->count) ->
               return_rack->count ) ;
    dispatch ( head(buf->r) -> return_itemtype ) ;
    wait for results ;
}
```

3) An important aspect of the design here is indicating the modification of the objects. For this we attach \( R \) at the method put since this method modified the p-manager. Thus, \( R[p\cdot rack] \) put is substituted in place of put. This modification of body will be implemented at the coding stage. The design of other constructs is similar and will not be discussed here.
Garbage collection for any unreferenced objects will be performed along the execution.

7 Discussion

In this paper, we have presented an architecture-independent approach to software development for parallel processing systems based on the computation model PROOF, which incorporates the functional paradigm into the object-oriented paradigm. The software will be developed in PROOF/L, which is a C++-based language with additional constructs required by PROOF. Then, PROOF/L code is translated to a target language of the given parallel processing system for execution. Our approach relieves the programmers from the consideration of processor topology and various parallelization aspects of the software development, and facilitates the exploitation of parallelism at both levels of granularity - object level and method level. The software developed using our approach will be easy to understand and modify since the structure of the software reflects the structure of the problem. Functional paradigm used in the method design also permit the expression of massive and implicit parallelism.

We have developed the front-end and back-end translators for the nCube and KSR-1 parallel processing systems and a number of examples have been experimented. We are incorporating the optimization techniques, such as function call in-lining, useless expression removal and common subexpression extraction, as well as implementing the grain size determination algorithm in order to improve the performance of the target codes. CASE tools need to be developed to aid the software developer in various stages in our approach, such as checking the consistency in the decomposition stage, design process and transformation of the design into the corresponding Petri-Net models.

Acknowledgement

We would like to acknowledge the participation of M. Chidambaram, G. Pour, V. Satish, W. Sung, M. Wells, and K. Yeom in this project, especially their implementation of this approach.

References


