WARASA: An Enhanced C++ for Concurrent Programming on Shared Memory Multiprocessor Computers

Yun JIANG, Akifumi MAKINOUCHI
Dept. of Computer Sci. & Comm. Eng., Kyushu University
Hakozaki, Higashi, Fukuoka 812, Japan.
E-mail: jiang@csce.kyushu-u.ac.jp, akifumi@csce.kyushu-u.ac.jp

Abstract

We propose a new object-oriented programming language called WARASA for writing concurrent object-oriented programmes on Mach-based shared memory multiprocessor computers. WARASA is an extension to C++ to allow programmers to develop concurrent programming.

In designing WARASA, we adopt a new way that combines the parallel mechanisms on Mach operating system with the object-oriented paradigm. It enables users to develop concurrent programmes easily and efficiently, because some implementation details (such as locking and unlocking) which programmers would have to know in order to write parallel programmes are hidden.

In this paper, we first show an overview on WARASA, and then, discuss the features of WARASA class models. Some applications are shown in order to demonstrate WARASA's expressive power.

1 Introduction

Thousands of low-cost commercial multiprocessor computers became available, and many computer scientists who are engaged in parallel programming have begun to pay attention to the development of software for multiprocessor computers. This is so, because the factor most commonly blamed for the slow transition to parallelism is lack of software support[1]. The present level of language support for parallel programming requires the user to expend more effort in managing the problem-solving resource, than in actually solving the problem. As a result, parallelism remains inaccessible or underutilized by most of the user community[2].

The concurrent object-oriented programming language that combines objects with parallelism is expected to reduce the complexity of parallel programming[5]. In fact, a certain number of concurrent object oriented programming languages have been developed, ABCL/I[4] and Concurrent Smalltalk[5] are some typical examples. However, there are some defects to be improved. For example, using Concurrent Smalltalk, it is necessary that the waiting object has to be restarted by passing a message run, furthermore using ABCL/I, users have to use a semaphore object to perform exclusion function.

In this paper, we describe a concurrent object-oriented programming language called WARASA. It is a "pure" object-oriented programming language, because it is an extension to C++[6] by adding capability of parallel programming.

One of the design objectives of WARASA is to hide implementation details which programmers would have to know in order to write parallel programmes using the functions supported by the Mach operating system. Another objective is to provide users with a high-level object-oriented abstraction for parallel programming.

WARASA depends on Mach operating system in both the concept and implementation of parallel programming. Mach is a multiprocessor operating system kernel providing capability-based interprocess communication. Based on the Mach operating system, two programming languages called Matchmaker[7] and Avalon/C++[8] have been developed. Matchmaker's goal is to provide language interfaces for object operation, and to support the distributed and object-oriented environment. It doesn't provide a "pure"object-oriented programming environment, unlike Smalltalk. Avalon/C++ uses the inheritance mechanism of C++ for handling operating-system-level details such as transaction management, internode communication, and automatic crash recovery. This idea is similar to ours. However, Avalon/C++ differs from WARASA in that its goal and design are to provide a reliable distributed system for helping the programmer to cope with the behavior associated with concurrency and failures[8].

WARASA runs on LUNA multiprocessor computers under the Mach operating system. The implementation of parallel mechanism on WARASA depends on the C Threads Package of Mach operating system.

We describe WARASA in the following sections. First, we will give an overview about WARASA, and then, three new kinds of classes will be discussed. These are the autonomous class, the exclusive class, and the synchronous class. WARASA's synchronization mechanisms will be discussed in section 3. In section 4, a parallel hash-join algorithm as an example of WARASA programmes will be shown in order to demonstrate WARASA's expressive power. Finally, in section 5, we will present the conclusion of this paper.

0730-3157/92 $3.00 © 1992 IEEE
2 WARASA

In this section, we describe the WARASA programming language in various aspects.

2.1 WARASA Overview

WARASA is a concurrent object-oriented programming language used for shared memory multiprocessors. An overview about WARASA will be presented in three respects.

2.1.1 WARASA as an Extension to C++

C++ has the important features that the object-oriented programming requires, such as data-abstraction and inheritance. The most important merit of C++ is that these features are available with a low runtime cost. It is worldwide emerging as the most widely used object-oriented language, because it is derived from the popular C programming language. Since it is considered as a successor to C, C programmers can enjoy its benefits[9]. That is the reason why we chose it, and extended it to support parallelism.

In designing WARASA, we extend C++ by adding a few new kinds of classes for writing parallel programmes. There are four kinds of basic classes provided for WARASA programming environment, viz. ordinary C++ classes, autonomous classes, exclusive classes and synchronous classes. The details of these classes are described in section 2.2.

2.1.2 WARASA Parallelism

WARASA is designed by combining the object-oriented programming language C++ with the parallel mechanism supported by the Mach operating system. The implementation is based on C Threads package provided by the Mach operating system.

Mach is a multiprocessor operating system, which supports five basic abstractions: Port, Message, Task, Memory Object, and Thread[10]. The Mach operating system splits the usual process notion into task and thread notions. A thread is the basic unit of CPU utilization, and the task is an execution environment in which more than one threads may run. When a programme in Mach is loaded, it holds one task and has the right to create threads within this task. The C Threads package allows programmers to control multiple threads providing shared variables, mutual exclusion for critical sections, and condition variables for synchronization of threads[10].

Although programmers can directly use the C Threads Package for developing their parallel programmes, they have to learn how to use it, which is a difficult and tedious work. In addition, as our experiments showed, it takes long time to correct errors in using the C Threads, if the users don't understand clearly what the C Threads Package's primitives mean and how they work.

By encapsulating the parallelism of C threads in objects, WARASA hides the functionality of C Threads Package from the users, so that they can write parallel programmes using concurrent object-oriented notions which are a direct extension to the C++ object orientation.

2.1.3 User's View

When a main programme written in WARASA is executed, it holds a thread of the Mach operation system. If the autonomous objects defined in the programme are created, these autonomous objects also run in parallel on the multiple threads. Exclusive objects are created, if they are concurrently shared by autonomous objects. Synchronous objects are used for synchronization among autonomous objects.

Since WARASA includes the C++ functionality, programmers may choose to operate an object, either parallel or sequential. If programmers want to define non-parallel operation, they may define ordinary C++ classes, and when the parallel execution is needed, they may choose autonomous classes whose operations can be executed in parallel.

2.2 WARASA Classes

As we already mentioned, the four kinds of classes in WARASA are ordinary class, autonomous class, exclusive class, and synchronous class[11]. Since the ordinary class is the same as the C++ class, no explanation about it will be given. In this section, the characteristics of the other three kinds are described using some examples.

2.2.1 Autonomous Class

Autonomous objects are the objects that can run in parallel on threads. An example of the autonomous class definition is shown below.

```
auto class producer { 
    private: 
        buffer* buf_OID; 
    public: 
        producer(buffer* buf_p); 
        "producer"(); 
    }
```

Keyword 'auto' meaning 'autonomous' distinguishes the autonomous class from other types. One of the important features of the class is that only two methods can be defined on it. One is the constructor and the other the distructor. In the example, 'producer(buffer* buf.p)' is the constructor and '"producer()"' is the distructor. The main function of constructor and distructor is to create an object and remove an object, respectively. However, some features different from the ordinary C++ class are indicated below.

i) Executing the constructor of an autonomous class results in the creation of an object of the class and the creation of a new thread on which the operation defined by the constructor runs.

ii) The main function of the autonomous class is designated by its constructor. Since the object is active as long as the thread is alive, the function is usually executed repeatedly.

iii) There are two states that an autonomous object holds, the execution state and the waiting state. Once an autonomous object is created, it remains in the execution state. The state remains same until it falls in the waiting state or it is destroyed. If the condition that the autonomous object waits for is satisfied, the autonomous object is relieved from the waiting state.
and is executed again automatically.

iv) The distructor is different from the one for the ordinary C++ class, in that it includes the removal of the thread.

2.2.2 Exclusive Class

It is known that in the shared memory multiprocessor systems, the semaphore technique for ensuring mutual exclusion is very important. In WARASA programmes, more than one autonomous object may send a message to an object at the same time.

In order to ensure that all the messages are processed correctly, the object has to receive and process one message at a time. That is, the object is exclusive in the sense that it is locked with exclusive mode while processing a message. Objects having this features are called exclusive objects. In WARASA, we call the class of these objects exclusive class. The variables defined in the exclusive class are shared by autonomous objects. The keyword 'excl' indicates the class to be defined as an exclusive class.

excl class < class_name >

Implementation of exclusive objects is straightforward. It uses the locking mechanism of C Threads, i.e., the mutual execution of shared variables type provided directly by the Mach operating system. A private variable of type 'mutex.t' is implicitly declared in each exclusive class to be defined. When a method defined in the class is provoked, the 'mutex.t' variable is locked and unlocked after the method is executed. These operations, including the variable declaration, are invisible to users.

2.2.3 Synchronous Class

A synchronous object encapsulates condition variables for synchronization. Its major function is to synchronize autonomous objects. A synchronous class is defined by adding the keyword 'sync' as follows.

class < class_name >

Figure 1. shows an example of synchronous class called buf_condition class, which is the abstraction of the availability of a buffer.

sync class buf_condition {
private:
condition_t rem_available; dep_available;
public:
buf_condition()
- buf_condition()
void wait_dep_available()
void wait_rem_available()
void make_dep_available()
void make_rem_available()
}

Figure 1. The Definition of a Synchronous Class

buf_condition

An important characteristic of the synchronous class definition is that, a synchronous class to be defined must include condition variables of type 'condition_t' such as rem_available and dep_available. These condition variables are used in the methods by which autonomous objects can be synchronized. The method wait_dep_available using the condition variable dep_available is shown below.

buf_condition::wait_dep_available()


c_condition_wait( dep_available );
return;

}

When the buffer is empty, and therefore not available, the above method is provoked by some producer autonomous object. It has to wait until the buffer becomes available. As a synchronous object is shared among multiple autonomous objects, the object may be accessed simultaneously via messages by the autonomous objects. Therefore, each synchronous object uses a lock mechanism similar to the one used for exclusive objects, and like the exclusive object, the locking and unlocking operations are hidden from the users.

In the next section, the implementation of the synchronous function is discussed in detail.

3 Synchronization Mechanisms

Exclusive and synchronous objects are used for cooperation and synchronization of autonomous objects. Since they are primitive, they can be combined to make high-level abstraction for cooperation and synchronization. In this section, how to use the primitive classes for various styles of synchronization mechanism is presented by using simple but typical examples.

3.1 Simple Style

The simple style uses the primitive synchronous mechanism of WARASA. The private data in a synchronous object includes only the condition variables whose type is defined by the system. The method defined for the synchronous object is the simple synchronization operation without any condition judgment. If it is used for implementing synchronization in an application, users have to design another object called the exclusive object in which shared variables are encapsulated, and both types of the objects must be used in pairs.

In our example, using the simple style, a synchronous class called 'buf_condition' class is defined in Figure 1 (shown in section 2.2.3). In this class, two condition variables are included. These are the 'rem_available' and 'dep_available'. Two pairs of synchronous operations are defined in four methods, in which the two condition variables are used. For example, one pair of the methods, 'wait_rem_available()' and 'make_rem_available()', uses the same condition variable 'rem_available'. When the method 'wait_rem_available()' is executed, the autonomous object sending the message will be stopped and enter into the waiting state. That is, the thread on which the object runs is suspended. When an autonomous object sends a message that calls the method 'make_rem_available()', the autonomous object that executed wait_rem_available() re-starts.

Since the 'buf_condition' synchronous class doesn't have any information concerning the available state of the buffer, we need to design an exclusive class called int_buffer. The class looks as follows.
excl class int_buffer {
  private:
    int *head;
    int *tail;
    int size;
  public:
    int_buffer(int size);
    ~int_buffer();
    int buf_deposit(int c);
    int buf_remove();
  }

Further, both the buf_condition object and the int_buffer object have to be used in pairs, so that the objects behave cooperatively with this shared buffer. For example, a consumer programme removing data from the buffer may be written in the following way:

```cpp
rem_d = buf_OID -> buf_remove();
while (rem_d == 0) {
  condition_OID -> wait_rem_available();
  rem_d = buf_OID -> buf_remove();
}
condition_OID -> make_dep_available();
```

The synchronization process of consumer and producer is depicted in Figure 2.

However, the disadvantage of this simple style is that actions of concerned objects are very primitive, so, relatively many messages have to be sent between the objects. In addition, overheads by locking/unlocking operations that are used in implementing exclusive and synchronous objects might be problematic. (Although users don't need to write explicitly these operations in their programmes.)

3.2 Mixed Style

The mixed style is proposed here to cope with the problems in the simple style. In order to reduce the messages to be passed as well as the locking/unlocking operations, we combine a simple synchronous object with a simple exclusive object to make a new complex synchronous object.

The private data of the synchronous object of this style are both condition variable and shared variable.

Therefore, the operations on the condition variables and the ones on shared variables are also mixed in order to be the operations for one object. If we rewrite the consumer and producer problem in this style, a synchronous class named buffer can be defined.

```cpp
sync class buffer {
  private:
    int *head;
    int *tail;
    int size;
  condition_t rem_available; dep_available;
  public:
    buffer(int size);
    ~buffer();
    int buf_deposit(int c);
    int buf_remove();
  }

//The method buf_remove() is shown.
buffer :: buf_remove(){
  int rm_data;
  if (head!=tail){
    rm_data=*tail;
    tail=load(tail--,size);
    condition_signal(dep_available)
    return rm_data;
  }
  else{
    condition_wait(rem_available);
    return 0;
  }
}
```

It can be seen that in the mixed style, synchronous operations are mixed with the operation on the shared variable. So, the consumer object that removes data from the buffer becomes very simple, i.e., it only issues a message 'rem_d= buf_OID -> buf_remove()' to the buffer object. Figure 3 depicts the consumer-producer process using the mixed style.

![Figure 2. A Synchronous Object in Simple Style](image)

![Figure 3. A Synchronous Object in Mixed Style](image)
3.3 Inheritance Style

The inheritance style is based on the inheritance of C++. The Mach's system-defined operations are encapsulated in a superclass, which in turn becomes a WARASA's system-defined class. When the user needs to define a synchronous class of his own, he or she doesn't need to know the Mach's system-defined operations. The only thing that he or she has to know is how to inherit from the synchronous superclass existing in WARASA.

Using the same problem, the inheritance style is explained. A class called conditionsync is used in superclass, which is a WARASA system defined class.

class condition_sync {
  private:
    condition_t sig1; sig2;
  protected:
    mutex_t Ylock-sig;
  public:
    condition_sync();
    ~condition_sync();
    void condi_wait_sig1();
    void condi_signal_sig1();
    void condi_wait_sig2();
    void condi_signal_sig2();
}

A mutex_t variable named Ylock-sig is defined by WARASA, which is not only in the condition_sync class, but also in all the methods of its derived classes. However, the description is hidden in the superclass definition. For example, a der.buffer class is defined as its derived class as follows.

sync class der_buffer: public condition_sync {
  private:
    int *head;
    int *tail;
    int size;
    condition_sync parent;
  public:
    int_buffer(int m2);
    ~int_buffer();
    int deposit (int c);
    int remove();
}

\% The remove method can be written in follows:
der_buffer :: remove( ){
  int rm_data;
  if (head==tail){
    rm_data=tail;
    tail=mod(tail--,size);
    condi_signal_sig2( );
    return rm_data; }
  else{
    condi_wait_sig1( );
    return 0; }
}

Here, note that the 'condition.wait_sig1()' is different from the 'condition.wait(rem.available) ' in the mixed style. The advantage in the inheritance style is that synchronous operations can be defined in the superclass provided by the WARASA system, so that programming becomes simple. However, it is difficult to provide superclasses that fit to all the synchronous applications the WARASA system allows. We need to go further for a more elegant style that would allow to describe naturally synchronous classes in WARASA.

4 Parallel Hash-Join Algorithm

The join operation is a time-consuming operation in the relational database system. In order to improve the performance of the join operation for the relational databases, many parallel hash-join algorithms have been provided.

The parallel hash-join algorithm discussed here is an extension to the GRACE Hash-Join algorithm([12][13]). The algorithm is composed of two phases, one is the partitioning-phase and the other is the joining-phase. Let R and S be the two relations to be joined. In the partitioning phase, hash-value of the join attribute of all tuples in both R and S relations are calculated, and then, these tuples are partitioned into different buckets, according to their hash values. When this operation is over, the next phase begins, where the tuples in each bucket are checked and joined if their join attributes are equal. In order to implement the parallel hash-join algorithm, we define six classes as shown in Table 1.

* main program

All the objects are created and deleted by the WARASA main programme. In our programme, four hash.join autonomous objects, and ten hash.value.bucket exclusive objects are created. Once the hash.join objects are created, each of them begins to calculate the hash values of the different tuples of relations R and S in parallel. It is to be noted that the parallel programmes are written with the ordinary syntax of C++ and without additional parallel operations, such as fork, join, lock, unlock, and so on.

* hash.join class

The hash.join class is defined as an autonomous class. Relations R and S are partitioned into 4 parts and each one is handled by a hash.join object. The followings are its main operations.

1. The hash values of tuples in a part of relations R and S are calculated, and then, according to their hash values, these tuples are partitioned into different hash.value.bucket objects.
2. The hash.join object passes a message to the join.sync object, in order to synchronize all the hash.join objects. That is so because any hash.join object must wait until others finish the operation in partitioning phase before the join phase begins.
3. The hash.join object asks the hash.value.bucket object to do the join operation, and then to form a new result relation in the hash Joined_tuples object. The operations described above are repeatedly executed, until all of the hash.value.bucket objects finish join operation.
The hash-value-bucket class and hash-joined-tuples class are defined as two exclusive classes. We design each hash-value-bucket object having two hash-tables with the same hash value range. One is used for relation R and the other is used for S. The private data in a hashjoined tuples object are new tuples which are obtained as a result of hash-join.

Here, we adopt the inheritance style version (see section 3.3) to provide a synchronization for hash-join objects. A class named condition_broa provided by WARASA system is used as a superclass. The joinsync class is a derived class of the condition_broa. The whole process of the parallel hash-join algorithm in WARASA is described in Figure 4.

![Figure 4. The Implementation of Parallel Hash Join Algorithm](image)

5 Summary and Conclusions

In this paper, we have described the design of WARASA, a concurrent object-oriented programming language on Mach-based shared memory multiprocessor computers.

Some differences between WARASA and the other concurrent object-oriented programming languages are the follows: i)WARASA combines the parallel mechanism of multiprocessor operation with the object-oriented paradigm, which allows users to develop parallel programmes easily and efficiently. ii)Since all the parallel operations of WARASA programmes are executed on multiple lightweight processes instead of processes, WARASA’s overhead is less than the other concurrent object-oriented programming languages. iv)WARASA introduces autonomous objects as entities of parallel execution, so that its ability of expressing parallel actions is better than that of the others.

The works to be done are to implement the WARASA preprocessor that translates programmes in WARASA into C++ programmes and to expand WARASA, so that it enables users to write distributed and parallel programmes using lightweight processes.

References