6 Object Oriented Parallel Programming Languages

6.1 Introduction

Concurrent object-oriented programming paradigm has been drawing great attention in wide areas, since concurrent objects provide a suitable abstraction for concurrent programs, and they are also units of concurrency to be distributed over multicomputers.

An essential motivation behind concurrent object-oriented programming is to exploit the software reuse potential of object-oriented features in the development of concurrent systems. Early attempts to introduce concurrency to object-oriented languages uncovered interferences between object-oriented and concurrency features that limited the extent to which the benefits of object-oriented programming could be realized for developing concurrent systems. This has fostered considerable research into languages and approaches aiming at a graceful integration of object-oriented and concurrent programming. In this chapter we will examine the issues underlying concurrent object-oriented programming and languages, examine and compare how different approaches for language design address these issues.

6.2 Parallel and concurrent programming

Concurrency or parallelism refer to the potentially parallel execution of parts of a computation. In a concurrent computation, the components of a program may be executed sequentially, or they may be executed in parallel. Concurrency provied us with the flexibility to interleave the execution of components of a program on a single processor, or to distribute it among several processors. Concurrency or parallelism abstract away some of the details in an execution, allowing us to concentrate on conceptual issues without having to be concerned with a particular order of execution which may result from the specialities of a given system. [GulAgha90]

Generally, the expression "concurrent programming" refers to system programming of single and multiprocessor machines with shared or distributed memory architecture. Good example for this are the modern operating systems. "Parallel programming" especially means creating special, parallel executable algorythms which speed up the execution of mathemathical computations which require extremely high power or extremely fast solution. Although, we may distinguish these two cases, both require similar models and tools.

6.3 Object oriented paradigm

The object oriented paradigm is a very modern and fashionable programming and software designing methodology. It is an especially suitable tool for building and maintaining large software systems.

Object oriented programming languages integrate a host of techniques that have proven useful for the development and maintenance of software and that promote software reusability. It is also expected that software development support systems will be more successful at developing applications by configuring pre-packaged software if they use object oriented techniques and are based on an object oriented programming language. Objects can be defined as entities which encapsulate data and operations into a single computational unit. Object oriented programming builds on the concepts of objects by supporting patterns of reuse and classification, for example through the use of inheritance which allows all instances of a particular class to share the same method.

Therefore, the main goals of object oriented paradigm are achieving high level of

1. data abstraction
2. data encapsulation
3. software modularity
4. code reuse

The object oriented development is based on the following main features:

- classes and instances
- inheritance
- polymorphism

Note: The "object oriented" and "object based" expressions mean similar but not the same paradigm. Object based languages does not provide inheritence which is a basic object oriented feature and very important in code reuse. It is because inheritence and synchronization constraints often conflict with each other.

6.4 Object orientation and parallelism

The property which initially suggests an easy match between the ideas of parallelism and object orientation is the remarkable similarity between the basic consctructs of both object orientation and concurrent systems. It is hard to miss the analogies between objects and processes, or more accurately between the underlying abstractions: classes and process types. Both categories of constructs support:

- Local variables (attributes of a class, variables of a process or process type)
- Persistent data, keeping their value between successive activations
- Encapsulated behavior (a single cycle for a process; any number of routines for a class)
- Heavy restrictions on how modules can exchange information
- A communication mechanism usually based on some form of message passing

The integration of object oriented and concurrent programming is promising for the development of software for a number of applications. However, the design of programming languages that keep up with this promise is a difficult task. The concurrent features of a language may interfere with its object oriented features making them hard to integrate in a single language or cause many of their benefits to be lost. For instance, encapsulation in sequential object oriented programming languages protects the internal state of objects from arbitrary manipulation and ensures its consistency. If concurrent execution is introduced in a language independently of objects it will compromise encapsulation, since concurrent execution of the operations of objects may violate the consistency of their internal state.

The approach taken for concurrency should also be carefully considered so that it is compatible with the principles underlying object oriented software development. A bad choice concerning the concurrency features could cause objects to be designed in a way that only fits the concurrency requirements of a particular application. Therefore it would be difficult to reuse these objects in different applications.

6.4.1 Object oriented parallel programming models

The approaches followed by OOPPLs vary considerably with respect to what objects stand for. Objects may be considered as processes, shared passive abstract data types, or as encapsulations of multiple processes and data.

In some languages objects do not have any predefined properties concerning concurrency. Objects are in general similar to those in sequential OOPLs. The concurrent features of the language may be used to implement objects that have some properties concerning concurrent execution. For instance, it is possible to implement objects that protect their internal state by synchronizing concurrent invocations of their operations, as it is also possible to have objects whose internal state is not protected from concurrent invocations of their operations. The language makes no distinction between these different kinds of objects. It is the responsibility of the programmers to design applications in such a way that operations of "unprotected" objects, will not be invoked concurrently.

On the other hand, other languages take the approach that all objects have some predefined properties concerning concurrency. For instance, in some languages objects are processes that communicate by exchanging messages, while in others objects have the property that execution of their operations is serialized.

There are three main categories of OOPPLs.

- The Orthogonal Approach: Concurrent execution is independent of objects. Synchronization constructs such as semaphores or monitors must be judiciously used for synchronizing concurrent invocations of object methods. In the absence of explicit synchronization, objects are subject to the activation of concurrent requests and their consistency may be violated.

- The Homogeneous Approach: All objects are considered to be "active" entities that have control over concurrent invocations. The receipt of request messages is delayed until the object is ready to service the request. There is a variety of constructs that can be used by an ob ject to indicate what method invocation it is willing to accept next.
- The Heterogenous Approach: Both active and passive objects are provided. Passive objects do not synchronize concurrent requests.

The latter two cases are also called non-orthogonal approaches.

In languages belonging to the non-orthogonal category, objects are associated some properties concerning concurrency. The property of objects that is common to all languages in this category, is that the internal state of objects is protected from concurrent execution of their operations. This property alleviates the problems that may occur with languages belonging to the orthogonal category if "unprotected" objects are used in a concurrent environment. This is accomplished in two ways: either the internal state of all objects is "protected" by default or there are objects of "protected" and "unprotected" kinds. In the latter case, the language distinguishes between these kinds of objects and disallows the use of "unprotected" objects in a context where their operations could be invoked concurrently.

Concurrent execution may be expressed by explicitly creating new threads of control, independently of objects, that communicate and synchronize by invoking the operations of shared objects. Another approach is to consider objects as active entities and express concurrent execution and synchronization by the creation of objects and their interaction. We use these two approaches to subdivide the uniform category into the categories integrated and non-integrated. For languages in the integrated category concurrent execution is expressed by interaction of objects whereas in the non-integrated category another concept like a process or activity is used for expressing concurrent execution.

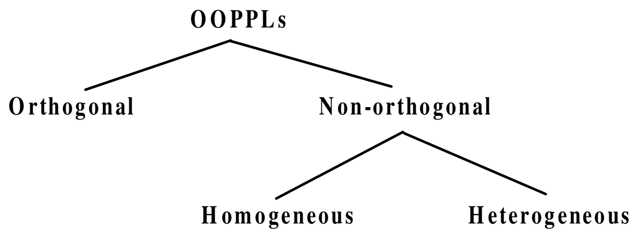

Figure 6-1.

From a reusability point of view languages in the orthogonal category have the disadvantage that in order to reuse objects one has to be aware whether their operations may be invoked concurrently. In order to function correctly objects depend on the environment in which they are used. When designing new objects special attention has to be paid for making them reusable for concurrent applications even if they were originally needed for a sequential application and are inherently sequential.

The non-orthogonal class has the advantage of preventing the problems that could occur by using "unprotected" objects in concurrent environment. Objects of sequential nature may be implemented in much the same way as in sequential languages. The mutually exclusive execution of the object's operations is handled automatically by the language.

The distinction between different kinds of objects characterizing the non-uniform category presents some disadvantages compared to the uniform one. The programmer has to decide in advance if a certain object should be of the "protected" or "unprotected" kind. Type hierarchies of "protected" and "unprotected" objects are typically kept disjoint, therefore objects having similar behavior may have to be defined twice introducing a certain redundancy in the class hierarchy. This approach may, on the other hand, have some performance gains. Unprotected objects may be implemented more efficiently, since invocations of their operations are not synchronized.

The integrated approach has the advantage that concurrent applications are structured in terms of objects which are the units of concurrent execution. The communication and synchronization of objects is expressed at the object interface which is clearly defined. In contrast to this approach, the non-integrated approach tends to organize applications in terms of processes that call shared passive objects for their communication and synchronization. This obscures the points of interaction of processes with their environment and makes them harder to reuse. [Ferenczi96]

6.4.2 Parallel programming

Independently of the classification of the previous section, a variety of approaches and notations are taken for concurrency.

Concurrent programming languages can be classified in the three categories: procedure-oriented, message-oriented and operation-oriented. The characteristic of procedure-oriented languages is that processes communicate through shared variables, synchronization constructs, as for instance monitors, are used for synchronization and mutual exclusion. The last two categories are based on message passing for communication and synchronization. The difference between these two categories is identified by the explicit use of primitives for sending and receiving messages in message-oriented languages, and structured implicit message passing, like remote procedure calls or rendez-vous, in operation-oriented languages.

Languages in each of these categories are roughly equivalent in expressive power. Languages in the procedure-oriented category are not suitable for distributed systems be cause of the cost of simulating shared memory when there is none. Operation-oriented and message-oriented languages may be implemented both on shared memory systems and distributed systems. Operation-oriented languages are better suited for programming client-server forms of process interaction while message-oriented are best suited for pipelined computations.

6.4.3 Process structure of objects

The process structure of objects determines the number of processes that may be active within an object at the same time, and how they are created and synchronized.

We may distinguish between static and dynamic process structure. A module has a static process structure if the number of threads that may be executing within a module is fixed (DSP like model). The programmer is responsible for multiplexing requests among this fixed number of threads. A module has a dynamic process structure if the number of processes that may be executing within a module is variable (DP like model). Processes are scheduled by the system and the programmer synchronizes access to shared variables. With this definition we could say that for example an Ada task is a module with static process structure whereas an Ada package is a module having dynamic process structure.

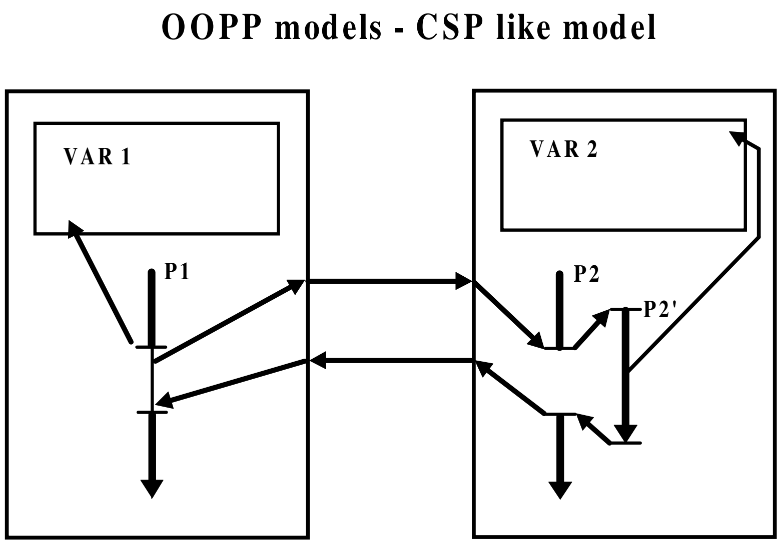

Figure 6-2.

Sometimes the term process is used for a concurrently executing object in a concurrent object based language. A thread is defined as what would be called a sequential process from an operating systems point of view. Processes are classified with respect to the properties of their threads into the categories:

- sequential
- quasi-concurrent
- concurrent

Sequential processes have a single thread of control, quasi-concurrent processes have several threads of control but only one thread may be executing at a time. Execution of threads in quasi-concurrent processes is multiplexed in a way similar to coroutines. Concurrent processes have several threads that may be executing concurrently.

We will say that an object has a dynamic process structure when a variable number of processes may be executing concurrently within an object and they are created automatically by an operation invocation. With this definition all languages in the orthogonal category support objects with dynamic process structure. For languages which support objects with a variable number of processes, but where the processes are not created automatically by operation invocations, we will say that they support objects with concurrent processes or more simply concurrent objects.

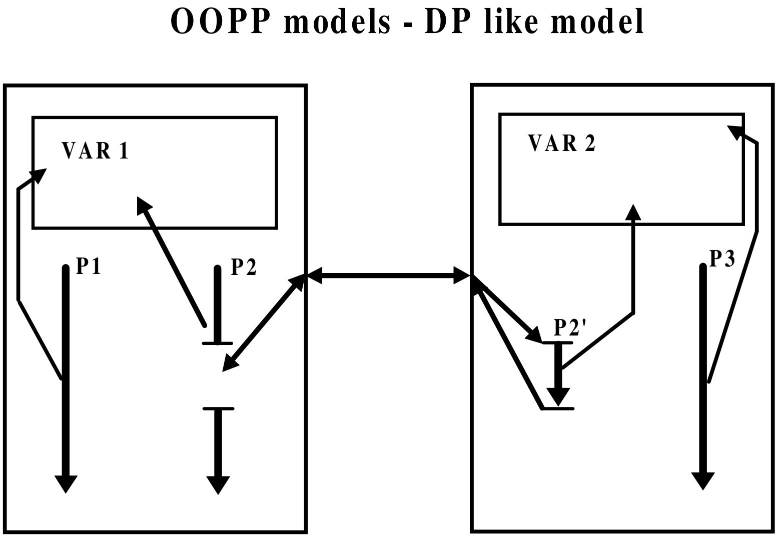

Figure 6-3.

We will use the terms single-process object and quasi-concurrent object in the place of sequential process and quasi-concurrent process, while we are going to use the term concurrent object for objects with concurrently executing threads that are not created automatically by operation invocations.

6.4.4 Object communication

In order to compare and contrast approaches to communication and synchronization, we view all concurrent object-oriented systems as collections of message-passing objects. By viewing the principal communication mechanism between objects as message passing we may focus on the interaction between objects playing client or server roles rather than on low-level issues con cerning the interleaving and scheduling of threads of control within an object system. This will allow us to consider more easily issues of reuse from the perspective of clients and servers. We may distinguish between languages providing one-way message passing primitives and those providing higher-level primitives that enforce balanced sending of request and reply messages.

- One-way Message Passing: the handling of request and reply messages must be explicitly programmed. Objects are not obliged to obey a request/reply protocol. Message passing may be synchronous or asynchronous.
- Request/Reply: communication primitives guarantee that requests will be eventually matched by replies. These primitives vary in the flexibility in sending and receiving messages they offer to clients and servers. Some approaches make use of "proxy" objects to disassociate the sending or receiving of messages from the current thread of control.

6.4.5 Concurrency requirements for objects

The main requirement is that modules should be able to turn their attention to another activity if the currently executing activity has to be delayed.

Two different situations can be identified in which this is needed: local delay and remote delay. The first situation arises when a server may not process immediately a client's request, for instance, because of the temporary unavailability of some local resource. In this situation the server should be able to put aside the current request and turn its attention to other requests that could be processed immediately. Remote delay is encountered when a server, in order to process a client's request, invokes another module. The module called by the server may not be able to process the request immediately or it could take a lot of time. In the meanwhile the server could accept requests from other clients.

Another reason for dealing with remote delays is for avoiding unnecessary deadlocks. In a system where inter-module communication paths are organized hierarchically a local delay in a lower level module produces problems similar to the problem of nested monitor calls.

Languages in the orthogonal category do not impose any a priori restrictions on the process structure of objects. This has the advantage that it is flexible for expressing solutions to synchronization problems and it makes it easy to implement objects concurrently.

The main disadvantage is that the programmer has to deal explicitly with mutual exclusion problems because of the dynamic process structure.

Languages in the non-orthogonal category are more restrictive with respect to the process structure of objects. The main objective of this approach is to make it easier to handle mutual exclusion problems. In some languages mutual exclusion problems are eliminated, in others one has to be concerned with mutual exclusion only when there is explicit interleaving of threads or when the creation of concurrent processes for executing the operations of objects is explicitly requested. On the other hand, depending on the restrictions imposed and the choice of communication and synchronization primitives it may become more difficult to deal with some concurrency problems.

6.4.6 The non-orthogonal solution

There are mainly three approaches with respect to the restrictions imposed on the process structure of objects in this category. The most restrictive one views objects as sequential processes. In this case no problems may occur because of access to shared variables since only one process has access to an object's instance variables. It is also easier to prove the correctness of operations since they are executed sequentially. We will name this approach single-process objects.

A second approach is to permit multiple threads of control to execute within an object in a quasi-concurrent fashion with only one thread that is active at a time. The interleaving of threads occurs in well defined points and it is controlled explicitly. We will name this approach quasi-concurrent objects.

The most permissive approach permits multiple threads to be active at the same time within an object. Taking this approach may require that additional synchronization primitives have to be introduced for controlling the interference of threads that share the instance variables of an object. We will say that this approach supports concurrent objects. It should be noted that the difference between this approach and the dynamic process structure of objects in the orthogonal category is that in the orthogonal category threads are created automatically at receipt of a message whereas in this case the object controls explicitly when and for which messages new threads are created.

6.4.7 Data abstraction

The benefit of data abstraction is the development of abstractions that more closely model physical or conceptual entities occurring in an application domain and promotes higher level programming. The effective realization of the abstract data type is hidden behind an interface. Operations defined at the interface capture the behavior of the entity modeled by the abstract data type and protect the internal state used to implement the abstract data type from inconsistent use. This separates the concerns of realizing the abstractions and programming by specifying interactions between the abstractions which model the entities of an application domain. The implementation of an abstract data type may be replaced, extended, without affecting applications making use of the abstract data type.

For evaluating which abstraction mechanisms are adequate for OOPPLs it is essential to identify what information should be captured and what should be hidden. There are three main ways how the objects relate to concurrency:

- An object's representation may need to be protected from concurrent invocation of its operation.
- The object is used to model a real world process or an entity with time varying behavior.
- The object is implemented in a concurrent fashion although the modeled entity may not be inherently concurrent.

The first and third points address issues that are more related with an object's representation. There are languages that ensure that the representation of objects is protected from concurrent execution. In these languages the object abstraction is powerful enough to hide this representation issue. The second point addresses the problem of objects with time varying behavior. Such objects are able to decide whether and when they will respond to incoming messages.

In most OOPPLs the behavior of such objects is expressed by using the concurrency constructs of the language. The design of concurrent constructs of languages vary in their ability to convey information about the abstract behavior without detailed examination of the representation, and with respect to their expressive power.

Support for mechanisms similar to data abstraction in OOPPLs is even more badly needed than in sequential languages but also is hard to provide. Ideally the data abstraction mechanism provided by OOPPLs should separate the aspects of concurrent execution relevant to an object's implementation from the concurrent behavior of the abstraction.

The OOPP languages in the non-orthogonal category extend the data abstraction mechanism of sequential languages since objects are associated with concurrent behavior.

6.5 Fundamental problems - inheritence anomalies

When OOPP languages are put to use in the development of large-scale programs, one of the prime issues is synchronization of activities of objects in the system. When a concurrent object is in a certain state, it can accept only a subset of its entire set of messages in order to maintain its internal integrity. We call such a restriction on acceptable messages the synchronization constraint of a concurrent object. In most OOPP languages, the programmer gives aither implicit or explicit program specification the synchronization specification. The synchronization specification must always be consistent with the synchronization constraint of an object; otherwise the object might accept a message which it really should not accept, causing an error. [MatsYone93][Ferenczi95]

Another important facility of object oriented languages id inheritance. However, it has been previously pointed out that inheritance and synchronization constraints often conflict with each other. Some have gone so far as not adopting inheritance in their languages (object based languages, eg. ABCL, Procol) or employed a flexible communication mechanism independent of the inheritance hierarchy.

For example, consider a bounded buffer with methods put() and get(), where put() stores an item in the buffer and get() removes the oldest one; then, the synchronization constraint is that one should not get() from a buffer whose state is empty, etc. The satisfaction of constraints is not achieved automatically; the user must somehow program the methods to implement the object behavior that satisfy the synchronization constraint. Synchronization code is the portion of the method code where such synchronization is controlled. Synchronization code of an object must always be consistent with its synchronization constraint; otherwise, semantical error could result.

In order to program synchronization code, a parallel OO language provides some primitives and/or general schemes for object-wise synchronization, such as guarded methods. We refer to the scheme for achieving object-wise synchronization useing those primitives in the language as the synchronization scheme of the language. Unfortunately, it has been pointed out that synchronization code cannot be effectively inherited without non-trivial class redefinitions. Inheritence anomaly breaks the encapsulation of classes more severely compared to sequential OO languages, because it is possible to create a general example where none of the parent methods can be inherited! One notable fact is that occurrence of inheritence anomaly depends on the synchronization scheme of the language; in other words, there are cases where redefinitions are required for one synchronization scheme while unnecessary in another.

There were several early attempts to provide robust synchronization code re-use in parallel OO languages. They can be largely classified into those based on guarded methods, and those based on accept set specification. The former assigns a boolean guard predicate to each method, and only messages that satisfy the guard can be accepted. The latter has an object specify the next set of messages to be accepted within its method code. There are three major categories of inheritance anomaly:

1. State partitioning of acceptable states (State Partitioning Anomaly)
2. History-only sensitivness of acceptable states
3. Modification of acceptable states (State Modification Anomaly)

6.5.1 State partitioning anomaly

State partitioning anomaly occures in the acceptable set based schemes. The following algorythm is a definition of class b-buf (bounded buffer) using a synchronization scheme called behavior abstraction.

```
Class b-buf: ACTOR
  int in, out, buf[SIZE];
behavior:
  empty = {put}; partial = {put, get};
  full = {get};
public:
  void b-buf() {
    in = out = 0; become empty;
  }
  void put(int item) {
    in++; /* store an item */
    if (in == out + size) become full;
    else become partial;
  }
  int get() {
    out++; /* remove an item */
    if (in == out) become empty;
    else become partial;
  }
}
```

The user specifies within its method code, the next set of methods that can be accepted by the object with the become statement followed by the symbolic name of the set. Addition of new methods in a subclass is handled by redefining the set to contain the name of the new method appropriately.

Unfortunately, naive acceptable set based schemes cannot cope with 'partitioning' of acceptable state: consider a class x-buf2, a subclass of b-buf. x-buf2 has an additional method get2(), which removes the two oldest items from the buffer simultaneously.

```
Class x-buf2: b-buf {
behavior:
  x_empty   = rename empty;
  x_one     = {put, get};
  x_partial = {put, get, get2()} redef partial;
  x_full    = {get, get2()} redef full;
public:
  void x-buf2() {
    in = out = 0;
    become x-empty;
  }
  intpair get2() {
    out += 2;
    if (in == out)        become x_empty;
    else if (in == out+1) become x_one;
    else            become x_partial;
  }
/* The following redefines the methods in b-buf */
  void put(int item) {
```

```
    in++; /* store an item */
    if (in == out)        become x_empty;
    else if (in == out+1) become x_one;
    else                  become x_partial;
  }
  int get() { /* requires a similar redefinition */
    ...
}
```

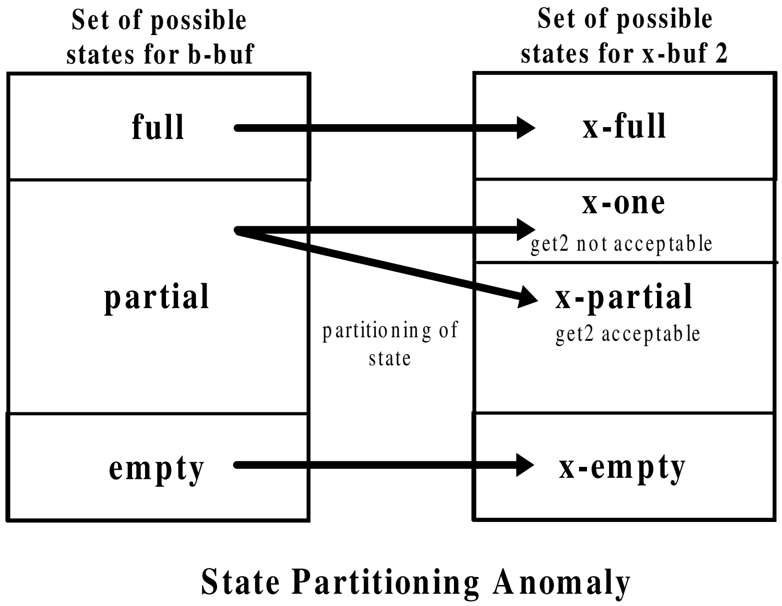

Figure 6-4.

The corresponding synchronization constraint for get2() requires at leasr two items exist. Thus the partial state must be partitioned into (1) x-one (only one item exists), and (2) x-partial (the remaining states). Then, all the non-initializer methods in b-buf (i.e., get and put) must be redfeined. This is due to the following reason: the set of possible 'states' an object can be partitioned into disjoint subsets, according to the synchronization constraint of the object. When a new method is added in the subclass, this partitioning may require further partitioning to account for the synchronization constraint of the new method (eg. get2() above). When this state partitioning is determined within the method code using explicit conditional statements as in the example, method redefinitions are required, because the new partitioning must be accounted for within all the methods. Note that, this is not entirely resolved by making accept sets first-class values as is with enable sets, because this partitioning cannot be affected by the operations upon the accept-set data.

6.5.2 History-only sensitiveness of acceptable states

State partitioning anomaly can be avoided with guarded methods, because they are able to directly judge whether the message is acceptable or not under a given state. Thus, even if the new methods were added, the guards would not need be redifined: for example, the guard (in >= out + 2) would precisely satisfy the synchronization constraint for get2(). However, other types of anomalies occur, as we illustrate below:

```
Class b-buf: ACTOR {
  int in, out, buf[SIZE];
public:
  void b-buf() { in = out = 0; }
  void put() when (in < out+size) { in++; }
  int get() when (in >= out+1) { out++; }
}
```

```
/* gb-buf is a subclass of b-buf with gget() */
Class gb-buf: b-buf {
  bool after-put;
public:
  void gb-buf() { after-put = False; }
  /* Definition of gget() */
  int gget() when (!after-put && (in >= out+1))
    { out++; after-put = True; }
  /* The following must be redefined */
  void put(int item) when (in < out+size)
    { in++; after-put = True; }
  int get() when (in >= out+1)
    { out++; afetr-put = False; }
}
```

Consider a subclass of b-buf, gb-buf that adds a single method, gget(). The behavior of gget() is almost identical to that of get(), with the exception is that it cannot be accepted immediately after the invocation of put(). A a consequence, both get() and put() must be redefined. The reason for the anomaly occurrence is that we cannot judge the state for accepting the gget() message with the guard declarations in b-buf, requiring addition of a flag variable after-put and the associated redefinition of guards. To be more specific, gget() is a history-only sensitive method. Since proper valuation of this variable must be done in all methods, the requirement of redefinitions of all parent methods arose.

6.5.3 State modification anomaly

We next consider the Lock class, an abstract mix-in class whose purpose is to be 'mixed-into' other classes in order to add the capability of locking an object. Upon accepting a message lock(), the object suspends the acceptance of further messages until it accepts unlock(). When Lock is mixed into b-buf to create lb-buf, it should not affect the method codes of b-buf since the state of the object with respect to lock() and unlock() is totally orthogonal to the effect of other messages. However, this is not the case - we must add an instance variable locked which indicates whether the object is currently 'locked' or 'unlocked'. then, the inherited methods such as put() or get() must be overridden in order to account for locked.

```
Class Lock: ACTOR {
  bool locked;
public:
  void Lock() {locked = 0;}
  void lock() when (!locked) {lock = 0;}
  void unlock() when (locked) {lock = 0;}
}
/* lb-buf is a subclass of b-buf with Lock mix-in */
Class lb-buf: b-buf, Lock {
public:
  void lb-buf();
  /* The following methods must be redefined */
  void put(int item)
      when (!locked && (in < out+size))
    { in++; }
  int get() when (!locked && (in >= out+1))
    { out++; }
}
```

What anomaly has occurred here? The execution of the methods in Lock modifies the set of states under which the methods inherited from the parent could be invoked. Thus, the mixing-in of lock to lb-buf introduces finer-grained distinction for the set of states under shich get() or put() can be invoked. This would require the modification of the method guards to account for the new synchronization constraint, resulting in state modification anomaly.

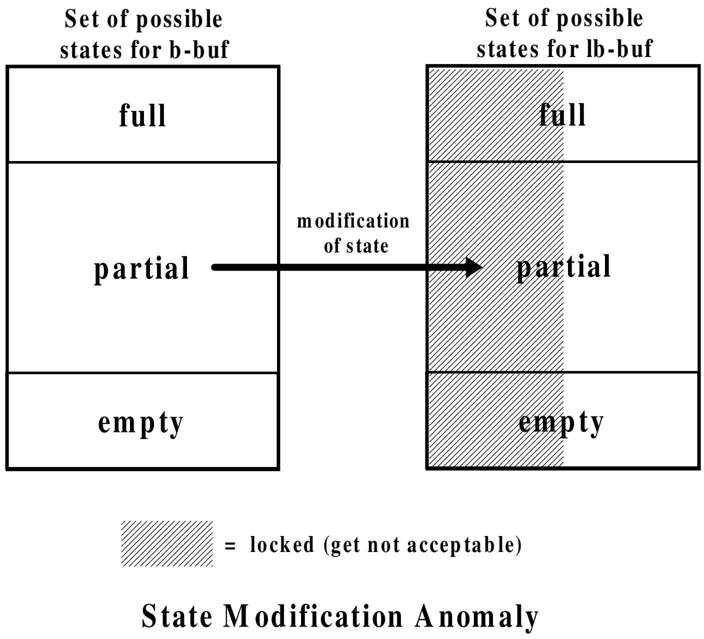

Figure 6-5.

6.6 Requirements for object oriented parallel programming languages

A great variety of programming notations have been used in concurrent programming languages. The main requirements on these notations were to express parallel execution and the synchronization and communication of concurrent threads. Most notations are roughly equivalent, providing adequate expressive power for common concurrency problems. However, not all of them are equally well-suited for the same kind of computations. Operation oriented languages, which are based on remote procedure call communication primitives, are best suited for computations where processes interact as clients and servers. Message oriented languages, supporting one-way message passing, are best suited for pipelined computations. Also, distributed and concurrent programs that are structured as a collection of active modules acting as servers and clients impose specific requirements on the combination of the concurrency features of a language.

Although various approaches have been followed for the design of parallel object-oriented languages not all of them are suitable for object-oriented programming. Object-oriented programming imposes additional requirements on the concurrency features of programming languages. The main requirement is that the concurrency features of a language be compatible with object-oriented features such as encapsulation, data-abstraction and inheritance, and promote the development of applications by combining and tailoring already existing objects:

- Encapsulation: object oriented parallel programming languages should be designed in such a way that encapsulation ensures the integrity of the internal state of objects even when their methods are invoked concurrently.
- Data abstraction: The concurrency constructs should make it possible for the clients to ignore the details of object implementations and for the implementors to be able to provide new compatible implementations without affecting the clients.This is necessary for supporting data abstraction and taking advantage of abstract typing for extensibility and code reuse.
- Inheritance: The design of the concurrency constructs of object oriented languages should take inheritance into account for providing constructs that combine well with class inheritance. The integration of inheritance and concurrency in object oriented languages is a difficult task.

- Promote object oriented software development: The concurrency constructs should be compatible with the client-server structure of object-oriented programs and facilitate the development of applications by reusing and adapting existing objects.

There are some additional important OOPP language design requirements:

- Minimality of mechanism: Object oriented software construction is a rich and powerful paradigm, which would intuitively seem to be ready for supporting concurrency. It is essential, then, to aim for the smallest possible extension. Minimalism here is not just a question of good language design. If the concurrent tension is not minimal, some concurrency constructs will be redundant with the object oriented constructs, or conflict with them, making the programmer's task difficult or impossible. To avoid such a situation, we must find the smallest syntactic and semantic epsilon that will give concurrent execution capabilities to our object oriented programs.
- Full use of available hardware parallelism: the OOPP language should allow the programmer to take advantage of the parallel capabilities of the hardware. The efficient use of the different existing HW architectures within one language is not easy. Without using the full power of the HW, parallel programming will lose one the key motivations.
- Support for reuse of nonconcurrent software: A criterion of the desirable category is the ability to reuse existing, nonconcurrent software, especially libraries of reusable software components. This may not always be achievable, since concurrency places new demands on the software structure. When existing software cannot be reused exactly "as is", the work involved in making it applivable to concurrent development should be reasonable, involving for example the writing of simple "wrapper" classes encapsulating existing sequential classes.

6.7 Introducing parallelism to object oriented systems

There have been numerous attempts in the recent years to combine parallel programming with object oriented programming. We can identify three distinct approaches for introducing concurrency to object-oriented systems:

1. *Design a new concurrent object oriented language*
   Most of the earlier systems took this approach: design a new OO language with built-in concurrency. These new languages provide powerful concurrency abstractions and general purpose programming capabilities. Some examples: ABCL, Pool, SR, Hybrid
2. *Extend an existing object oriented language*
   Most of the extensions introduce concurrency to their respective languages using some combination of the following techniques:
   - inheritence from special concurrency classes that the modified compiler recognizes (eg.: Eiffel, pC++)
   - special keywords, modofiers or preprocessing techniques to modify or extend the language syntax and semantics (eg.: uC++, CC++, CEiffel)
   - extension to the syntax and semantics of the language to support a general concurrency paradigm such as the Actor model (eg.: ACT++)
3. *Design a library.*
   Use an existing object oriented language and provide concurrency abstractions through external libraries. The library based solutions are attractive, since they do not replace the existing software development platform. However, the sequential execution semantics of the host language may impose restrictions on providing type-safety and intraobject concurrency. (eg. Eiffel Classes, Java)

Most of the existing and the most widely used OOPP languages are derivated from C++, or are extensions to C++. No doubt, the reason of this is the extrem popularity of C++. C++ is the most known object oriented language. Unfortunately it is far not the ideal language for object oriented programming, but because it derived from the well known, wide spread, portable C, many programmers could easily switch to it. Some of the parallel C++ languages: CC++, pC++, MPC++, UC++, EC++, ICC++, HPC++, MC++, Mentat, Charm++, Para++, etc.)

In the rest of this chapter we will examine the details of two of the OOPP languages. First is the CC++ (Compositional C++). It is derived from C++ and it is widely used. The second choice is the Java with the concurrency libraries. Java is a relatively new language which is developing and spreading very fast and which has some remarkable, unique features among the object oriented languages.

6.8 The CC++ language

Compositional C++ (CC++) is a small set of extensions to C++ for parallel programming. CC++ provides constructs for specifying concurrent execution, for managing locality, and for communication. It allows parallel programs to be developed from simpler components using sequential, parallel, and concurrent composition.

6.8.1 Introduction

CC++ is a general-purpose parallel programming language comprising all of C++ plus six new keywords. It is a strict superset of the C++ language in that any valid C or C++ program that does not use a CC++ keyword is also a valid CC++ program. The CC++ extensions implement six basic abstractions:

1. The processor object is a mechanism for controlling locality. A computation may comprise one or more processor objects. Within a processor object, sequential C++ code can execute without modification. In particular, it can access local data structures. The keyword global identifies a processor object class, and the predefined class proc_t controls processor object placement.
2. The global pointer, identified by the type modifier global, is a mechanism for linking together processor objects. A global pointer must be used to access a data structure or to perform computation (using a remote procedure call, or RPC) in another processor object.
3. The thread is a mechanism for specifying concurrent execution. Threads are created independently from processor objects, and more than one thread can execute in a processor object. The par, parfor, and spawn statements create threads.
4. The sync variable, specified by the type modifier sync, is used to synchronize thread execution.
5. The atomic function, specified by the keyword atomic, is a mechanism used to control the interleaving of threads executing in the same processor object.
6. Transfer functions, with predefined type CCVoid, allow arbitrary data structures to be transferred between processor objects as arguments to remote procedure calls.

These abstractions provide the basic mechanisms required to specify concurrency, locality, communication, and mapping.

6.8.2 Parallelism

A CC++ program, like a C++ program, executes initially as a single thread of control (task). However, a CC++ program can use par, parfor, and spawn constructs to create additional threads. A parallel block is distinguished from an ordinary C++ block by the keyword par, as follows.

```
par {
       statement1;
       statement2;
       ...
       statementN;
     }
```

A parallel block can include any legal CC++ statement except for variable declarations and statements that result in nonlocal changes in the flow of control, such as return.

Statements in a parallel block execute concurrently. For example, the following parallel block creates three concurrent threads: two workers and one master.

```
par {
   worker();
   worker();
   master();
}
```

A parallel block terminates when all its constituent statements terminate; execution then proceeds to the next executable statement. Thus, in the preceding parallel block, the thread that executed the parallel block proceeds to the next statement only when both the master and the workers have terminated.

A parallel for-loop creates multiple threads, all executing the same statements contained in the body of the for-loop. It is identical in form to the for-loop except that the keyword parfor replaces for. For example, the following code creates ten threads of control, each executing the function myprocess.

```
parfor (int i=0; i<10; i++) {
   myprocess(i);
}
```

Only the loop body of the parfor executes in parallel. Evaluation of the initialization, test, and update components of the statement follows normal sequential ordering. If the initialization section uses a locally declared variable (for example, int i), then each instance of the loop body has its own private copy of that variable.

CC++ parallel constructs can be nested arbitrarily. Hence, the following code creates ten worker threads and one master.

```
par {
   master();
   parfor (int i=0; i<10; i++)
      worker(i);
}
```

Finally, the spawn statement can be used to specify unstructured parallelism. This statement can be applied to a function to create a completely independent thread of control. The parent thread does not wait for the new thread to terminate execution, and cannot receive a return value from the called function.

6.8.3 Locality

In CC++ the concepts address space and thread of control are separated. Processor objects represent address spaces, and threads represent threads of control. Processor objects can exist independently of threads, and more than one thread can be mapped to a processor object.

6.8.3.1 Processor objects

A processor object is defined by a C++ class declaration modified by the keyword global. A processor object is identical to a normal C++ class definition in all but two respects:

1. Names of C++ ``global'' variables and functions (that is, names with file scope) refer to unique objects within different instances of a processor object. Hence, there is no sharing between processor object instances.
2. Private members of a processor object need not be explicitly declared to be private. C++ ``global'' functions and variables are defined implicitly to be private members of the processor object in which they occur.

Processor object types can be inherited, and the usual C++ protection mechanisms apply, so private functions and data are accessible only from a processor object's member functions or from the member functions of derived objects. Hence, it is the member functions and data declared public that represent the processor object's interface.

6.8.3.2 Global pointers

A processor object is a unit of locality, that is, an address space within which data accesses are regarded as local and hence cheap. A thread executing in a processor object can access data structures defined or allocated within that processor object directly, by using ordinary C++ pointers.

Processor objects are linked together using global pointers. A global pointer is like an ordinary C++ pointer except that it can refer to other processor objects or to data structures contained within other processor objects. It represents data that are potentially nonlocal and hence more expensive to access than data referenced by ordinary C++ pointers.

A global pointer is distinguished by the keyword global. For example:

```
float *global gpf;    // global pointer to a float
char * *global gppc; // global pointer to pointer of type char
C *global gpC;        // global pointer to an object of type C
```

When the new statement is used to create an instance of a processor object, it returns a global pointer. For example, the statement

```
Construction *global item = new Construction;
```

creates a new processor object of type Construction and defines item to be a pointer to that object.

6.8.3.3 Thread placement

By default, a CC++ thread executes in the same processor object as its parent. Computation is placed in another processor object via an RPC. A thread needs only a global pointer to another processor object to be able to invoke any of its public member functions. For example, in the following line bridge_pobj is a global pointer to the processor object on which the consumer is to execute, and bridge is a public member function of that object.

```
bridge_pobj->bridge();
```

A single thread executing in a processor object implements a task. Many CC++ programs create exactly one thread per processor object, yielding computation structures.

6.8.4 Communication

CC++ does not provide low-level primitives for directly sending and receiving data between threads. Instead, threads communicate by operating on shared data structures. For example, one thread may append items to a shared list structure, from which another thread removes items; this implements a form of channel communication. CC++ mechanisms can be used to implement a wide variety of such communication structures.

6.8.4.1 Remote operations

CC++ global pointers are used in the same way as C++ local pointers; the only difference is that we use them to operate on data or to invoke functions that may be located in other processor objects. Hence, the following code fragment first assigns to and then reads from the remote location referenced by the global pointer gp.

```
global int *gp;
int len2;
*gp = 5;
len2 = (*gp) * 2;
```

These read and write operations result in communication.

If we invoke a member function of an object referenced by a global pointer, we perform what is called a remote procedure call (RPC). An RPC has the general form

```
<type> *global gp;
result = gp->p(...)
```

where gp is a global pointer of an arbitrary <type>, p(...) is a call to a function defined in the object referenced by that global pointer, and result is a variable that will be set to the value returned by p(...). An RPC proceeds in three stages:

1. The arguments to the function p(...) are packed into a message, communicated to the remote processor object, and unpacked. The calling thread suspends execution.
2. A new thread is created in the remote processor object to execute the called function.
3. Upon termination of the remote function, the function return value is transferred back to the calling thread, which resumes execution.

Basic integer types (char, short, int, long, and the unsigned variants of these), floats, doubles, and global pointers can be transferred as RPC arguments or return values without any user intervention.

6.8.4.2 Synchronization

A producer thread can use an RPC to move data to a processor object in which a consumer thread is executing, hence effecting communication. However, we also require a mechanism for synchronizing the execution of these two threads, so that the consumer does not read the data before it is communicated by the producer. CC++ uses the sync variable. A sync variable is identified by the type modifier sync, which indicates that the variable has the following properties:

1. It initially has a special value, ``undefined."
2. It can be assigned a value at most once, and once assigned is treated as a constant (ANSI C and C++ const).
3. An attempt to read an undefined variable causes the thread that performs the read to block until the variable is assigned a value.

We might think of a sync variable as an empty box with its interior coated with glue; an object cannot be removed once it has been placed inside.

Any regular C++ type can be declared sync, as can a CC++ global pointer. Hence, we can write the following.

```
sync int i;          // i is a sync integer
sync int *j;         // j is a pointer to a sync integer
int *sync k;         // k is a sync pointer to an integer
sync int *sync l;    // l is a sync pointer to a sync integer
```

We use the following code fragment to illustrate the use of sync variables. This code makes two concurrent RPCs: one to read the variable length and one to write that variable.

```
Length *global lp;
int val;
par {
   val = lp->read_len();
   lp->write_len(42);
}
```

What is the value of the variable val at the end of the parallel block? Because the read and write operations are not synchronized, the value is not known. If the read operation executes before the write, val will have some arbitrary value. (The Length class does not initialize the variable length.) If the execution order is reversed, val will have the value 42.

6.8.4.3 Mutual exclusion

The sync variable allows us to synchronize the transfer of data from a producer to a consumer. In other situations, we may wish to allow two threads to operate on the same nonsync data structure while ensuring that they do not interfere with each other's execution. For example, the enqueue and dequeue operations allow a single sender and receiver to communicate by enqueuing to and dequeuing from a shared queue. What if we want multiple senders to be able to append messages to the same queue? We cannot allow two producers to make concurrent calls to enqueue, as an arbitrary interleaving of two enqueue calls could have bizarre results. What we need is a mechanism to ensure that only one message can be enqueued at a time.

This requirement is satisfied by CC++ 's atomic keyword. Member functions of an object can be declared atomic. This declaration specifies that the execution of such a function will not be interleaved with the execution of any other atomic function of the same object. For example, to allow multiple producers to append to the same queue, we would declare the enqueue function to be atomic, as follows.

```
atomic void Queue::enqueue(int msg) {
   tail->next  = new IntQData;
   tail->value = msg;
   tail        = tail->next;
}
```

This ensures that even if multiple producers attempt to append to the same queue concurrently, the actual enqueue operations will occur in some sequential order and a valid queue will be generated.

6.8.4.4 Data transfer functions

In C++ , declarations of the form

```
ostream& operator<<(ostream&, const TYPE& obj_in);
istream& operator>>(istream&, TYPE& obj_out);
```

in the class ios of the iostream library define infix operators << and >>, which can be used to write and read data of a specified TYPE to and from files. These operators are predefined for simple types and can be provided by the programmer for more complex types. This facility enhances modularity by allowing a class definition to specify how its data structures should be read and written. A program can then read and write instances of that class without being aware of their internal structure.

CC++ uses an analogous mechanism for communicating data structures between processor objects. Associated with every CC++ datatype is a pair of data transfer functions that define how to transfer that type to another processor object. The function

```
CCVoid& operator<<(CCVoid&, const TYPE& obj_in);
```

defines how TYPE should be packaged for communication. It is called automatically by the compiler whenever an object of TYPE needs to be transferred to another processor object, that is, whenever an RPC call takes an argument of that type or returns a value of that type. Similarly, the function

```
CCVoid& operator>>(CCVoid&, TYPE& obj_out);
```

defines how TYPE should be unpackaged. It is called by the compiler whenever an object of TYPE is received from another processor object. Upon termination of this call, obj_out will be a copy of the obj_in used as the argument to the operator << in the initial processor object.

The type CCVoid is a compiler-defined type analogous to the types istream and ostream used in the iostream library. Data transfer functions are generated automatically by the CC++ compiler for simple data types, but must be constructed by the programmer for local pointers, arrays, and structures that contain local pointers.

6.8.5 Asynchronous communication

The need for asynchronous communication can arise when the tasks involved in a computation must access elements of a shared data structure in an unstructured manner. This requirement can be satisfied in CC++ in three different ways:

1. The shared data structure can be encapsulated in a set of specialized data tasks to which read and write requests are directed by using channel operations.
2. The shared data structure can be distributed among the computation tasks. Each computation task must then poll periodically for pending requests.
3. A third implementation approach exploits CC++ 's RPC mechanism more directly. The shared data structure is distributed among the computation tasks. However, rather than sending a message on a channel, a task accesses data in another processor object by making an RPC to an appropriate member function.

6.8.6 Determinism

Determinism can greatly simplify program development. CC++ does not provide any guarantees of deterministic execution: indeed, the basic execution model is highly nondeterministic, allowing as it does the interleaved execution of multiple threads in a single address space. Nevertheless, there are simple rules that, if followed, allow us to avoid unwanted deterministic interactions. In particular, a CC++ program is easily shown to be deterministic if it uses a task-based concurrency model (one thread per processor object) and if tasks interact only by using the channel library with one sender and one receiver per channel.

While a task/channel model ensures determinism, there are also circumstances in which it is advantageous to use CC++ constructs in more flexible ways. For example:

1. Concurrent threads provide a mechanism for overlapping computation and communication. When one thread is suspended waiting for a communication, another thread can be executing. For example, the following code can perform computation while waiting for the remote datum, value.

```
par {
   value = pobj->get_remote_value();
   perform_computation();
}
use_remote_value(value);
```

2. RPCs that read and write data structures in other processor objects can be used to implement a variety of asynchronous communication mechanisms;
3. On a shared-memory computer, threads created in the same processor object can execute in parallel (on different processors), communicating by reading and writing shared data rather than sending and receiving data. This shared-memory programming model can improve performance relative to channel-based communication by reducing data movement and copying.

These more general forms of concurrency and communication introduce the possibility of complex, nondeterministic interactions between concurrently executing threads. However, the risk of nondeterministic interactions can be reduced substantially by avoiding the use of global variables, by making shared data structures have the sync attribute, and by ensuring that accesses to nonsync shared data structures occur within atomic functions.

6.8.7 Mapping

A parallel program defined in terms of CC++ constructs can be executed on both uniprocessor and multiprocessor computers. In the latter case, a complete program must also specify how the processor objects created by a CC++ program are mapped to processors.

Mapping in CC++ is a two-stage process. First, threads are mapped to processor objects, and then processor objects are mapped to processors. The mapping of threads to processor objects can be one-to-one, in which case it is the mapping of processor objects to physical processors that is important. Alternatively, the mapping of processor objects to physical processors may be one-to-one, in which case it is the mapping of threads to processor objects that is important. If both mappings are one-to-one, then the mapping problem is straightforward.

An important aspect of the second mapping stage, processor object placement, is that it influences performance but not correctness. Hence, we can develop a program on a uniprocessor and then tune performance on a parallel computer by changing placement decisions.

6.8.7.1 Processor object placement

By default, a newly created processor object is placed on the same processor as its creator. An alternative placement can be specified by using the placement argument to the new operator. In C++ , this argument is used to position an object in memory space; in CC++ , it can also be used to position a processor object in processor space. (It can also be used to specify where in a file system to find the code for a processor object; however, we do not discuss this facility here.) The location is specified by an implementation-dependent class named proc_t. The constructor functions proc_t and node_t defined in the CC++ library can be used to construct a placement structure with a specified processor name. These are used in the following code fragment, which creates a new processor object (of type MyClass) on a processor called mymachine.

```
MyClass *global G;
proc_t location(node_t("mymachine"));
G = new (location) MyClass;
```

The new statement creates a new processor object; the supplied proc_t object (location) specifies the machine name. To place the new processor object on a different processor, one need change only the second line of this code fragment, for example to the following.

```
proc_t location(node_t("yourmachine"));
```

As a further example, the following code creates 32 processor objects, placing each on a different processor of a multicomputer with nodes named sp#0, sp#1, ..., sp#31. Notice how parfor is used to create the different processor objects concurrently.

```
MyClass *global G[32];
parfor (int i=0; i<31; i++) {
   char node_name[256];
   sprintf(node_name,"sp#%
   proc_t location(node_t(node_name));
   G[i] = new (location) MyClass;
}
```

Although simple, this code represents bad programming practice, in that it embeds information about the environment in which the program is executing. A better approach is to encapsulate mapping decisions in a separate class. This class encapsulates two private variables (P and proc_names) that represent the environment in which a program is to execute. The member function initmap is used to initialize these variables. Two additional member functions, processor and random_p, return a proc_t object representing the ith processor and a randomly-selected processor, respectively. Finally, two data transfer functions (omitted for brevity) package and unpackage the node list associated with a mapping object, allowing a mapping to be passed as an argument when creating a new processor object.

6.8.7.2 Mapping threads to processor objects

An alternative approach to mapping in CC++ is to create a fixed number of processor objects onto which threads are then placed. This approach is often used in SPMD computations, in which case a single thread is mapped to each processor object. Another important application is in situations where a computation creates a large number of lightweight threads that interact only via global pointers. We can map these threads to a static number of processor objects, hence avoiding the overhead of creating a new processor object when creating a new thread; as the threads do not share local data structures, the mapping of threads to processor objects does not influence the result computed.

6.8.8 Performance

CC++ programs do not explicitly send and receive messages, but instead perform read and write operations on global pointers; make remote procedure calls; and use par, parfor, and spawn statements to create new threads of control. Nevertheless, the communication operations associated with a CC++ program can usually be determined easily. Normally, a read operation on a global pointer, a write operation on a global pointer, or an RPC all result in two communications: one to send the remote request and one to receive an acknowledgment or result. As noted in Chapter 2, the cost of each message can be specified with reasonable accuracy in terms of a startup cost and a per-word cost. It is necessary to distinguish between the communication costs incurred when communicating CC++ processes are located on different processors (interprocessor communication) or on the same processor (intraprocessor communication). Both these costs can depend significantly on implementation technology.

The following issues must also be considered when examining the performance of CC++ programs.

Reading and writing global pointers. Reading or writing a global pointer normally involves two communications: one to send the read or write request, and one to return a result and/or signal completion. Hence, global pointers must be used with caution, particularly on computers where communication is expensive. If a data structure is referenced often, it may be worthwhile to move that data structure to where it is used most often, or to replicate it. If a task requires many data items from the same processor object, it may be better to use an RPC to transfer all the required data in a single message.

Remote procedure calls. An RPC normally involves two communications: the first to transmit the procedure call and its data to the remote processor, and the second to signal completion and to return any result. In many situations, the return message is not required and hence represents overhead. This overhead can be avoided by using the spawn statement to create an asynchronous thread of control. For example, the following code sends a value on a channel,

```
void send(int val) { inport->enqueue(val); }
```

can be improved in cases where one does not care when the send operation completes, by rewriting it to eliminate the reply, as follows.

```
void send(int val) { spawn inport->enqueue(val); }
```

Fairness. When two or more threads execute in the same processor object, CC++ guarantees that execution is fair: that is, that no thread that is not blocked waiting for data will be prevented indefinitely from executing. However, the time that a thread waits before executing can vary significantly depending on characteristics of both the application and a particular CC++ implementation. Hence, care must be taken if application performance depends on obtaining timely responses to remote requests.

Remote operations. As a general principle, operations involving global objects (processor object creation, RPC, etc.) are more expensive than operations involving only local objects. However, the cost of these operations can vary significantly from machine to machine. An RPC is typically less expensive than a processor object creation, and a remote read or write operation is typically less expensive than an RPC. The first processor object creation on a processor is often significantly more expensive than subsequent processor object creation operations on the same processor.

Compiler optimization. Because CC++ is a programming language rather than a library, a compiler may in some situations be able to reduce communication costs by eliminating replies, coalescing messages, or otherwise reorganizing a program to improve performance.

6.9 Parallel programming, using Java

Java is a relatively new, high level, general purpose object oriented programming language which allows writing platform independent applications and supports networked environment especially well. It is similar to C++ (it is also derived from C) but it is more simple and it realizes the object oriented concepts in a very clear, natural way.

Using the "Synchronization" object library, concurrent applications can be written in Java. Also, JavaPVM is a Java interface for the well known PVM system, and it allows writing parallel, PVM based applications in Java language.

The third new and interesting experiment for using Java for parallel and concurrent applications or distributed computing is the so called Voyager system. Voyager is a Java agent-enhanced Object Request Broker (ORB). Voyager allows Java programmers to easily create network applications using agent-enhanced distributed programming techniques. It uses the regular Java message syntax to construct remote objects, sends them messages and moves them between applications. It can create autonomous agents that can roam on the network and continue to execute as they move. [ObjectSpace97]

We don't introduce the Java language here but we describe how the most important concurrent synchronization and communication methods can be constructed in Java, using the Synchronization object library. [CampWalr97]

6.9.1 Threads

A process is an executing program. It has been allocated memory by the operating system. A thread is an execution or flow of control in the address space of a process; the program counter register points to the next instruction to be executed. A process is a program with one thread. A process can have more than one thread. All the threads in a process have their own program counter and their own stack for local (also called automatic) variables and return addresses of invoked procedures.

In Java, a thread in the run-time interpreter calls the main() method of the class on the java command line. Each object created can have one or more threads, all sharing access to the data fields of the object.

Motivations of concurrent programming with threads:

- shared memory multiprocessors are cheaper and more common so each thread can be allocated a CPU;
- it is less expensive and more efficient to create several threads in one process that share data than to create several processes that share data;
- IO on slow devices like networks, terminals, and disks can be done in one thread while another thread does useful computation in parallel;
- multiple threads can handle the events (e.g., mouse clicks) in multiple windows in the windowing system on a workstation;
- in a LAN cluster of workstations or in a distributed operating system environment, a server running on one machine can spawn a thread to handle an incoming request in parallel with the main thread continuing to accept additional incoming requests.

In Java, the basic construct for concurrent programming is the parallal executed threads.

6.9.2 Semaphores

Semaphores can be used for mutual exclusion and thread synchronization. Instead of busy waiting and wasting CPU cycles, a thread can block on a semaphore (the operating system removes the thread from the CPU scheduling or ``ready'' queue) if it must wait to enter its critical section or if the resource it wants is not available.

Mutual exclusion pseudocode:

```
semaphore S = 1; ... P(S); N=N+1; V(S);
```

Condition synchronization pseudocode (resource availability):

```
semaphore tapeDrives = 7; ... P(tapeDrives); useTapeDrive();
V(tapeDrives);
```

Java has implicit binary semaphores of the form

```
Object mutex = new Object();
   /*...*/
synchronized (mutex) {
   /*...*/
}
```

that can be used for mutual exclusion. Only one thread at a time can be executing inside the synchronized block.

Java does not have explicit binary and counting semaphores, so they are provided as classes in the Synchronization subdirectory of the lib directory.

The bounded buffer algorythm implemented with semaphores:

```
import Utilities.*;
import Synchronization.*;

class BoundedBuffer extends MyObject {
                    // designed for a single
                    // producer thread and a single consumer thread
   private int numSlots = 0;
   private double[] buffer = null;
   private int putIn = 0, takeOut = 0;
   private int count = 0;
   private BinarySemaphore mutex = null;
   private CountingSemaphore elements = null;
   private CountingSemaphore spaces = null;

   public BoundedBuffer(int numSlots) {
      super("BoundedBuffer with " + numSlots + " slots");
      if (numSlots <= 0)
         throw new IllegalArgumentException("numSlots<=0");
      this.numSlots = numSlots;
      buffer = new double[numSlots];
      mutex = new BinarySemaphore(1);
      elements = new CountingSemaphore(0);
      spaces = new CountingSemaphore(numSlots);
      System.out.println( "BoundedBuffer alive,
                          numSlots=" + numSlots);
   }

   public void deposit(double value) {
      P(spaces);
      buffer[putIn] = value;
      putIn = (putIn + 1) % numSlots;
      P(mutex);
      count++;
      System.out.println(" after deposit, count=" + count
         + ", putIn=" + putIn);
      V(mutex);
      V(elements);
   }

   public double fetch() {
      double value;
      P(elements);
      value = buffer[takeOut];
      takeOut = (takeOut + 1) % numSlots;
      P(mutex);
      count--;
      System.out.println(" after fetch, count=" + count
         + ", takeOut=" + takeOut);
```

```
      V(mutex);
      V(spaces);
      return value;
   }
}
```

6.9.3 Monitors

Java uses the synchronized keyword to indicate that only one thread at a time can be executing in this or any other synchronized method of the object representing the monitor. A thread can call wait() to block and leave the monitor until a notify() or notifyAll() places the thread back in the ready queue to resume execution inside the monitor when scheduled. A thread that has been sent a signal is not guaranteed to be the next thread executing inside the monitor compared to one that is blocked on a call to one of the monitor's synchronized methods. Also, it is not guaranteed that the thread that has been waiting the longest will be the one woken up with a notify(); an arbitrary thead is chosen by the JVM. Finally, when a notifyAll() is called to move all waiting threads back into the ready queue, the first thread to get back into the monitor is not necessarily the one that has been waiting the longest.

Each Java monitor has a single nameless anonymous condition variable on which a thread can wait() or signal one waiting thread with notify() or signal all waiting threads with notifyAll(). This nameless condition variable corresponds to a lock on the object that must be obtained whenever a thread calls a synchronized method in the object. Only inside a synchronized method may wait(), notify(), and notifyAll() be called.

Methods that are static can also be synchronized. There is a lock associated with the class that must be obtained when a static synchronized method is called.

Usually all the publicly accessible methods, the service or access methods, will be synchronized. But a Java monitor may be designed with some methods synchronized and some not. The non-synchronized methods may form the public access and call the synchronized methods, which will be private.

The bounded buffer algorythm with monitor:

```
class BoundedBuffer {       // designed for a single producer thread
                            // and a single consumer thread
   private int numSlots = 0;
   private double[] buffer = null;
   private int putIn = 0, takeOut = 0;
   private int count = 0;

   public BoundedBuffer(int numSlots) {
      if (numSlots <= 0)
         throw new IllegalArgumentException("numSlots<=0");
      this.numSlots = numSlots;
      buffer = new double[numSlots];
      System.out.println( "BoundedBuffer alive,
                          numSlots=" + numSlots);
   }

   public synchronized void deposit(double value) {
      while (count == numSlots)
         try {
            wait();
         } catch (InterruptedException e) {
            System.err.println("interrupted out of wait");
         }
      buffer[putIn] = value;
      putIn = (putIn + 1) % numSlots;
      count++;                    // wake up the consumer
      if (count == 1) notify();   // since it might be waiting
      System.out.println(" after deposit, count=" + count
         + ", putIn=" + putIn);
   }
```

```
   public synchronized double fetch() {
      double value;
      while (count == 0)
         try {
            wait();
         } catch (InterruptedException e) {
            System.err.println("interrupted out of wait");
         }
      value = buffer[takeOut];
      takeOut = (takeOut + 1) % numSlots;
      count--;                           // wake up the producer
      if (count == numSlots-1) notify(); // since it might be waiting
      System.out.println(" after fetch, count=" + count
         + ", takeOut=" + takeOut);
      return value;
   }
}
```

6.9.4 Message passing

Sometimes the phrase "send a message to an object" is used to describe a thread in one object calling a method in another object. Here, that phrase will be used to describe a thread in one object sending a message to a thread in another object, where the message is itself an object.

This technique is used for thread communication and synchronization in a computing environment where the threads do not have shared memory (since the threads reside in different virtual or physical machines). Hence the threads cannot share semaphores or monitors and cannot use shared variables to communicate. Message passing can still be used, of course, in a shared memory platform.

Messages are sent through a port or channel with an operation like send(port, message) and received from a port or channel with an operation like receive(port, message). Messages can be passed synchronously, meaning the sender blocks until the received does a receive and the receiver blocks until the sender does a send. Since the sender and receiver are at specific known points in their code at a known specific instant of time, synchronous message passing is also called a simple rendezvous with a one-way flow of information from the sender to the receiver.

In asynchronous message passing, the sender does not block. If there is not a receiver waiting to receive the message, the message is queued or buffered. The receiver still blocks if there is no queued or buffered message when a receive is executed.

In conditional message passing, the message remains queued until some condition, specified by the receiver, becomes true. At that time, the message is passed to the receiver, unblocking it.

A two-way flow of information, perhaps over the network, is called an extended rendezvous and can be implemented with a pair of sends and receives. Typically a client thread will use this technique to communicate with a server thread and request a service to be performed on its behalf. A similar situation is a worker thread contacting a master thread, asking for more work to do.

- client or worker: send request; receive reply
- server or master: receive request; perform service; send reply

Messages are objects and can be

- passed by reference within the same Java Virtual Machine,
- or serialized through a pipe within the same JVM,
- or serialized through a socket between JVMs that are on the same physical machine or on different physical machines.

The base data types, int, double, etc., can be sent as messages in binary or raw data format through a pipe or socket using the DataInputStream and DataOutputStream methods. They can also be sent as objects using the wrapper classes Integer, Double, etc.

Synchronization Package Classes

Here is a collection of Java message passing classes. All of the message passing port classes implement the methods in the MessagePassing interface or the ConditionalMessagePassing interface. This exception is thrown when an error occurs. This exception is used in implementing restricted rights ports (below). All classes except the conditional ones extend this base class.

- Synchronous Port.
- Asynchronous Port. A Vector is used to queue sent but not yet received messages.
- Asynchronous Conditional Port. The receiver must pass an object that implements the Condition interface, that is the object must contain a checkCondition() method that is used to determine which messages sent are eligible to be received.
- Synchronous Conditional Port.
- Finite Buffer Asynchronous Port.
- Receive-Only Rights Port. Send-Only Rights Port. These two filter classes can be wrapped around a message passing port to permit only sending or receiving on the port. This is done by overriding the restricted method with one that throws NotImplementedMethodException.
- Serialized Objects as Messages in a Pipe or Socket Port. The objects are serialized and deserialized using the writeObject() and readObject() methods through a pipe within the same JVM or a socket between different JVMs.

The bounded buffer producer and consumer implemented with a set of empty messages representing the buffer slots.

```
import Utilities.*;
import Synchronization.*;

class Buffer { public String who; public double value;
               public long when;
   public Buffer() {who = null; value = 0.0; when = 0;}
   public String toString()
      {return " who="+ who + " value=" + value + " when=" + when;}
}

class Producer extends MyObject implements Runnable {

   private int pNap = 0; // milliseconds
   private MessagePassing mpEmpty = null, mpFull = null;

   public Producer(String name, int pNap,
         MessagePassing mpEmpty, MessagePassing mpFull) {
      super(name);
      this.pNap = pNap;
      this.mpEmpty = mpEmpty;
      this.mpFull = mpFull;
      new Thread(this).start();
   }

   public void run() {
      int napping;  double value;
      while (true) {
         napping = 1 + (int) random(pNap);
         System.out.println("age=" + age() + ", " + getName()
            + " napping for " + napping + " ms");
         nap(napping);
         value = random();
         System.out.println("age=" + age() + ", " + getName()
```

```
            + " produced value " + value);
         Buffer buffer = (Buffer) receive(mpEmpty);
         buffer.who = getName(); buffer.value = value;
         buffer.when = age();
         send(mpFull, buffer);
         System.out.println("age=" + age() + ", " + getName()
            + " deposited value " + value);
      }
   }
}

class Consumer extends MyObject implements Runnable {

   private int cNap = 0; // milliseconds
   private MessagePassing mpEmpty = null, mpFull = null;

   public Consumer(String name, int cNap,
         MessagePassing mpEmpty, MessagePassing mpFull) {
      super(name);
      this.cNap = cNap;
      this.mpEmpty = mpEmpty;
      this.mpFull = mpFull;
      new Thread(this).start();
   }

   public void run() {
      int napping;
      while (true) {
         napping = 1 + (int) random(cNap);
         System.out.println("age=" + age() + ", " + getName()
            + " napping for " + napping + " ms");
         nap(napping);
         System.out.println("age=" + age() + ", " + getName()
            + " wants to consume");
         Buffer buffer = (Buffer) receive(mpFull);
         System.out.println("age=" + age() + ", " + getName()
            + " fetched buffer" + buffer);
         buffer.who = null; buffer.value = 0.0; buffer.when = 0;
         send(mpEmpty, buffer);
      }
   }
}

class ProducerConsumer extends MyObject {

   public static void main(String[] args) {

      // parse command line arguments, if any, to override defaults
      GetOpt go = new GetOpt(args, "Us:p:c:R:");
      go.optErr = true;
      String usage = "Usage: -s numSlots -p pNap -c cNap -R runTime";
      int ch = -1;
      int numSlots = 20;
      int pNap = 3;       // defaults
      int cNap = 2;       // in
      int runTime = 60;   // seconds
      while ((ch = go.getopt()) != go.optEOF) {
         if      ((char)ch == 'U') {
            System.out.println(usage);  System.exit(0);
         }
```

```
         else if ((char)ch == 's')
            numSlots = go.processArg(go.optArgGet(), numSlots);
         else if ((char)ch == 'p')
            pNap = go.processArg(go.optArgGet(), pNap);
         else if ((char)ch == 'c')
            cNap = go.processArg(go.optArgGet(), cNap);
         else if ((char)ch == 'R')
            runTime = go.processArg(go.optArgGet(), runTime);
         else {
            System.err.println(usage);  System.exit(1);
         }
      }
      System.out.println("ProducerConsumer"
         + ", numSlots=" + numSlots
         + ", pNap=" + pNap + ", cNap=" + cNap
         + ", runTime=" + runTime);

      // create the kind of message passing
      MessagePassing mpEmpty = new AsyncMessagePassing();
      MessagePassing mpEmptyS = new MessagePassingSendOnly(mpEmpty);
      MessagePassing mpEmptyR = new MessagePassingReceiveOnly(mpEmpty);
      MessagePassing mpFull = new AsyncMessagePassing();
      MessagePassing mpFullS = new MessagePassingSendOnly(mpFull);
      MessagePassing mpFullR = new MessagePassingReceiveOnly(mpFull);

      // create the Producer and Consumer
      // (they start their own threads)
      new Producer("PRODUCER", pNap*1000, mpEmptyR, mpFullS);
      new Consumer("Consumer", cNap*1000, mpEmptyS, mpFullR);
      System.out.println("All threads started");

      // set up the bounded buffer with numSlots empty messages
      for (int i = 0; i < numSlots; i++) send(mpEmpty, new Buffer());

      // let them run for a while
      nap(runTime*1000);
      System.out.println("age()=" + age()
         + ", time to stop the threads and exit");
      System.exit(0);
   }
}
```

6.9.5 Remote Method Invocation (RMI)

Sun Microsystems has added a remote method invocation capability to Java, the ability to make remote procedure calls. The latter term we used above to describe an extended rendezvous between two threads in different JVMs, perhaps on different physical machines. Sun's RMI allows a thread in one JVM to invoke (call) a method in an object in another JVM that is perhaps on a different physical machine. A new thread is created in the other (remote) JVM to execute the called method. The following example shows how to use it. The ComputeServer remote object implements a Compute interface containing a compute() method that a local Client can call, passing a Work object whose doWork() method the server will call. The client is using the remote server to have work performed on its behalf (adding vectors). Presumably the server is running on a computer architecture that can perform the work more efficiently. Parameters to the remote method and the method's return result, if any, are passed from one JVM to the other using object serialization over the network.