Parallel Object Programming C++ User and Installation Manual

The POP-C++ Team
Grid and Ubiquitous Computing Group
http://gridgroup.hefr.ch

Software Version 1.3
Manual Version 1.3-a

University of Applied Sciences
of Western Switzerland, Fribourg

Parallel Object Programming C++
User and Installation Manual
Manual version: 1.3-a

This work was partially funded by the CoreGRID Network of Excellence, in the European Commission's 6th Framework Program.

The POP-C++ Team

Tuan Anh Nguyen
Pierre Kuonen
Marcelo Pasin
Jean-François Roche
Laurent Winkler

Contents

CHAPTER

1 Introduction and Background

1.1 Introduction

Programming large heterogenous distributed environments such as GRID or P2P infrastructures is a challenging task. This statement remains true even if we consider researches that have focused on enabling these types of infrastructures for scientific computing such as resource management and discovery [4, 6, 2], service architecture [5], security [14] and data management [1, 12]. Efforts to port traditional programming tools such as MPI [3, 11, 7] or BSP [13, 15], also had some success. These tools allow programmers to run their existing parallel applications on large heterogenous distributed environments. However, efficient exploitation of performance regarding the heterogeneity still needs to be manually controlled and tuned by programmers.

POP-C++ is an implementation, as an extension of the C++ programming language [8], of the POP (**P**arallel **O**bject **P**rograming) model first introduced by Dr. Tuan Anh Nguyen in his PhD thesis [9]. The POP model is based on the very simple idea that objects are suitable structures to distribute data and executable codes over heterogeneous distributed hardware and to make them interact between each other.

Inspired by CORBA [10] and C++, the POP-C++ programming language extends C++ by adding a new type of **parallel object**, allowing to run C++ objects in distributed environments. With POP-C++, programming efficents distributed applications is as simple as writing a C++ programs.

1.2 The POP Model

The POP model extends the traditional object oriented programming model by adding the minimum necessary functionality to allow for an easy development of coarse grain distributed high performance applications. When the object oriented paradigm has unified the concept of module and type to create the new concept of **class**, the POP model unifies the concept of class with the concept of **task** (or **process**). This is realized by adding to traditional sequential classes a new type of class: **the parallel class**. By instantiating parallel classes we are able to create a new category of objects we will call **parallel objects** in the rest of this document.

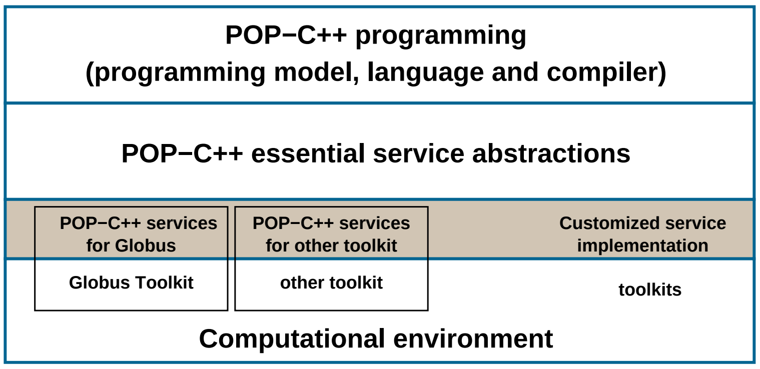
Parallel objects are objects that can be remotely executed. They coexist and cooperate with traditional sequential objects during the application execution. Parallel objects keep advantages of object-orientation such as data encapsulation, inheritance and polymorphism and adds new properties to objects such as:

- Distributed shareable objects
- Dynamic and transparent object allocation
- Various method invocation semantics

1.3 System Overview

Although the POP-C++ programming system focuses on an object-oriented programming model, it also includes a runtime system which provides the necessary services to run POP-C++ applications over distributed environements. An overview of the POP-C++ system architecture is illustrated in figure 1.1.

Figure 1.1 POP-C++ system architecture


POP−C++ programming
(programming model, language and compiler)
POP−C++ essential service abstractions
POP−C++ services for Globus
POP−C++ services for other toolkit
Customized service implementation
Globus Toolkit
other toolkit
toolkits
Computational environment

The POP-C++ runtime system consists of three layers: the service layer, the POP-C++ service abstractions layer, and the programming layer. The service layer is built to interface with lower level toolkits (e.g. Globus) and the operating system. The essential service abstraction layer provides an abstract interface for the programming layer. On top of the architecture is the programming layer, which provides necessary support for developing distributed object-oriented applications. More details of the POP-C++ runtime layers are given in a separate document [9].

1.4 Structure of this Manual

This manual has five chapters, including this introduction. The second chapter explains the POP-C+'s programming model. The third chapter describes the POP-C++ programming syntax. The fourth chapter explains how to compile and run POP-C++ applications. The fifth chapter shows how to compile and install the POP-C++ tool. Programmers interested in using POP-C++ should read first chapters 2, 3 and 4. System managers should read first chapter 5, and eventually chapters 2 and 4.

1.5 Additional information

More information can be found on the POP-C++ wiki web site which contains :

- A quick tutorial to get started with POP-C++
- Solutions to commonly found problems
- Programming examples
- Latest sources

`http://gridgroup.hefr.ch/popc`

CHAPTER

2 Parallel Object Model

2.1 Introduction

Object-oriented programming provides high level abstractions for software engineering. In addition, the nature of objects makes them ideal structures to distribute data and executable codes over heterogeneous distributed hardware and to make them interact between each other. Nevertheless, two questions remain:

- Question 1: which objects should run remotely?
- Question 2: where does each remote object live?

The answers, of course, depend on what these objects do and how they interact with each other and with the outside world. In other words, we need to know the communication and the computation requirements of objects. The parallel object model presented in this chapter provides an object-oriented approach for requirement-driven high performance applications in a distributed heterogeneous environment.

2.2 Parallel Object Model

POP stands for *Parallel Object Programming*, and POP parallel objects are generalizations of traditional sequential objects. POP-C++ is an extension of C++ that implements the POP model. POP-C++ instantiates parallel objects transparently and dynamically, assigning suitable resources to objects. POP-C++ also offers various mechanisms to specify different ways to do method invocations. Parallel objects have all the properties of traditional objects plus the following ones:

- Parallel objects are shareable. References to parallel objects can be passed to any other parallel object. This property is described in section 2.3.
- Syntactically, invocations on parallel objects are identical to invocations on traditional sequential objects. However, parallel objects support various method invocation semantics: synchronous or asynchronous, and sequential, mutex or concurrent. These semantics are explained in section 2.4.

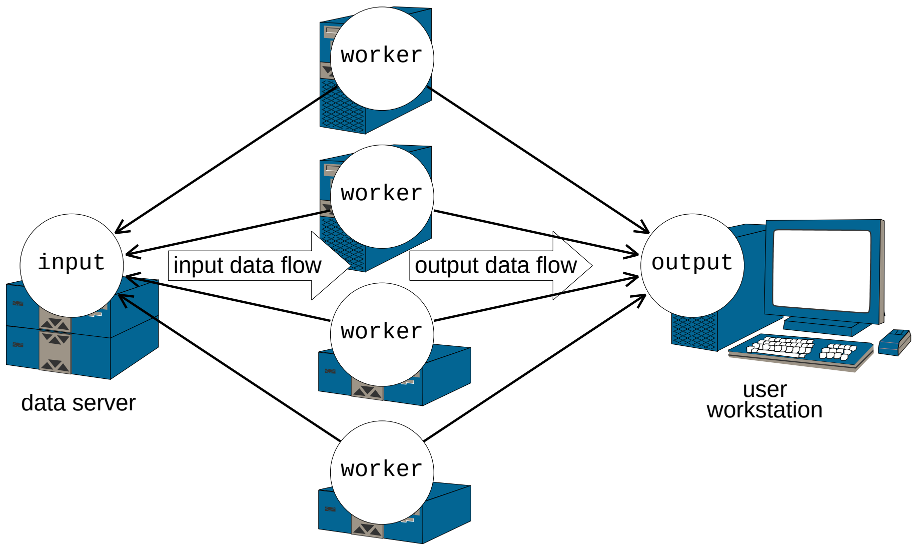
- Parallel objects can be located on remote resources in separate address spaces. Parallel objects allocations are transparent to the programmer. The object allocation is presented in section 2.5.
- Each parallel object has the ability to dynamically describe its resource requirement during its lifetime. This feature is discussed in detail in section 2.6

As for traditional objects, parallel objects are active only when they execute a method (non active object semantic). Therefore, communication between parallel objects are realized thank to remote methods invocation.

2.3 Shareable Parallel Objects

Parallel objects are shareable. This means that the reference of a parallel object can be shared by several other parallel objects. Sharing references of parallel objects are useful in many cases. For example, figure 2.1 illustrates a scenario of using shared parallel objects: `input` and `output` parallel objects are shareable among `worker` objects. A `worker` gets work units from `input` which is located on the data server, performs the computation and stores the results in the `output` located at the user workstation. The results from different `worker` objects can be automatically synthesized and visualized inside `output`.

Figure 2.1 A scenario using shared parallel objects



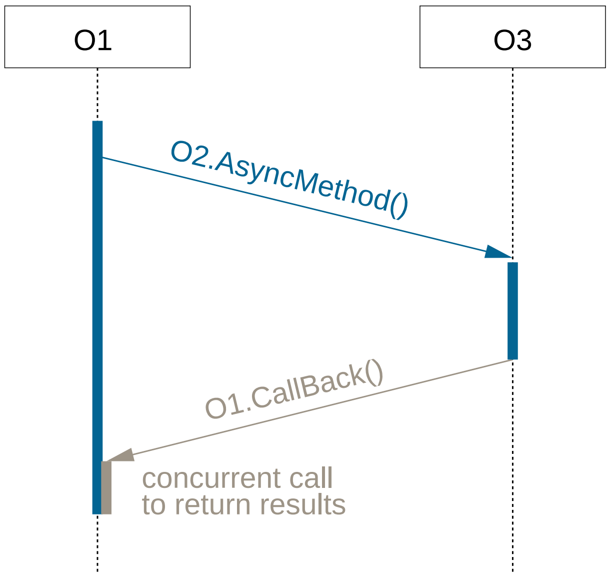
To share the reference of a parallel object, POP-C++ allows parallel objects to be arbitrarily passed from one place to another as arguments of method invocations.

2.4 Invocation Semantics

Syntactically, method invocations on parallel objects are identical to those on traditional sequential objects. However, to each method of a parallel object, one can associate different invocation semantics. Invocation semantics are specified by programmers when declaring methods of parallel objects. These semantics define different behaviours for the execution of the method as described below:

- **Interface semantics**, the semantics that affect the caller of the method:
  - **Synchronous invocation**: the caller waits until the execution of the called method on the remote object is terminated. This corresponds to the traditional method invocation.
  - **Asynchronous invocation**: the invocation returns immediately after sending the request to the remote object. Asynchronous invocation is important to exploit the parallelism. However, as the caller does not wait the end of the execution of the called method, no computing result is available. This excludes asynchronous invocations from producing results. Results can be actively returned to the caller object using a callback the caller. To do so the called object must have a reference to the caller object. This reference can be passed as an argument to the called method (see figure 2.2).

Figure 2.2 Callback method returning values from an asynchronous call

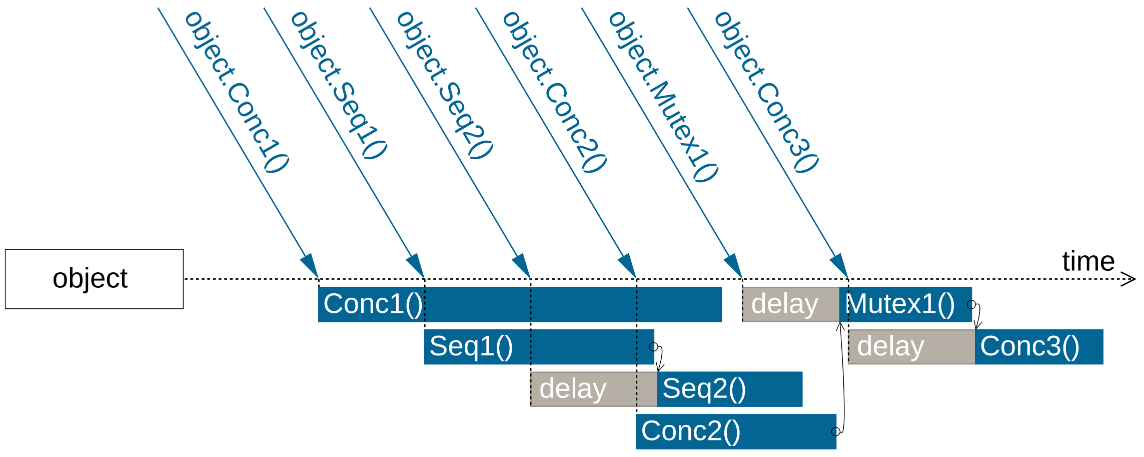


- **Object-side semantics**, the semantics that affect the order of the execution of methods in the called parallel object:
  - **A mutex call** is executed after completion of all calls previously arrived.
  - **A sequential call** is executed after completion of all sequential and mutex calls previously arrived
  - **A concurrent call** can be executed concurrently (time sharing) with other concurrent or sequential calls, except if mutex calls are pending or executing. In the later case he is executed after completion of all mutex calls previously arrived.

In a nutshell, different object-side invocation semantics can be expressed in terms of atomicity and execution order. The mutex invocation semantics guarantees the global order and the atomicity of all method calls. The sequential invocation semantics guarantees only the execution order of sequential methods. Concurrent invocation semantics guarantees neither the order nor the atomicity.

Figure 2.3 illustrates different method invocation semantics. Sequential invocation `Seq1()` is served immediately, running concurrently with `Conc1()`. Although the sequential invocation `Seq2()` arrives before the concurrent invocation `Conc2()`, it is delayed due to the current execution of `Seq1()` (no order between concurrent and sequential invocations). When the mutex

Figure 2.3 Example of different invocation requests



invocation `Mutex1()` arrives, it has to wait for other running methods to finish. During this waiting, it also blocks other invocation requests arriving afterward (`Conc3()`) until the mutex invocation request completes its execution (atomicity and barrier).

2.5 Parallel Object Allocation

The first step to allocate a new object is the selection of an adequate placeholder. The second step is the object creation itself. Similarly, when an object is no longer in use, it must be destroyed in order to release the resources it is occupying in its placeholder. The POP-C++ runtime system provides automatic placeholder selection, object allocation, and object destruction. This automatic features result in a dynamic usage of computational resources and gives to the applications the ability to adapt to changes in both the environment and the user behavior.

The creation of POP-C++ parallel objects is driven by high-level requirements on the resources where the object should lie (see section 2.6). If the programmer specifies these requirements they are taken into account by the runtime system for the transparent object allocation. The allocation process consists of three phases: first, the system finds a suitable resource, where the object will lie; then the object code is transmitted and executed on that resource; and finally, the corresponding interface is created and connected to the object.

2.6 Requirement-driven parallel objects

Parallel processing is increasingly being done using distributed systems, with a strong tendency towards web and global computing. Efficiently extract high performance from highly heterogeneous and dynamic distributed environments is a challenge today. POP-C++ was conceived under the belief that for such environments, high performance can only be obtained if the two following conditions are satisfied:

- The application should be able to adapt to the environment;
- The programming environment should somehow enables objects to describe their resource requirements.

The application adaptation to the environment can be fulfilled by multilevel parallelism, dynamic utilization of resources or adaptive task size partitioning. One solution is to dynamically create parallel objects on demand.

Resource requirements can be expressed by the quality of service that objects require from the environment. Most of the systems offering quality of service focus on low-level aspects, such as network bandwidth reservation or real-time scheduling. POP-C++ integrates the programmer requirements into parallel objects in the form of high-level resource descriptions. Each parallel object is associated with an object description that depicts the characteristics of the resources needed to execute the object. The resource requirements in object descriptions are expressed in terms of:

- Resource (host) name (low level description, mainly used to develop system services).
- The maximum computing power that the object needs (expressed in MFlops).
- The maximum amount of memory that the parallel object consumes.
- The expected communication bandwidth and latency.

An object description can contain several items. Each item corresponds to a type of characteristics of the desired resource. The item is classified into two types: strict item and non-strict item. A strict item means that the designated requirement must be fully satisfied. If no satisfying resource is available, the allocation of parallel object fails. Non-strict items, on the other hand, give the system more freedom in selecting a resource. Resource that partially match the requirements are acceptable although a full qualification resource is preferable. For example, a certain object has a preferred performance 150MFlops although 100MFlops is acceptable (non-strict item), but it need memory storage of at least 128MB (strict item).

The construction of object descriptions occurs during the parallel object creation. The programmer can provide an object description to each object constructor. The object descriptions can be parametrized by the arguments of the constructor. Object descriptions are used by the runtime system to select an appropriate resource for the object.

It can occur that, due to some changes on the object data or some increase of the computation demand, an object description needs to be re-adjusted during the life time of the parallel object. If the new requirement exceeds some threshold, the adjustment could cause the object migration. The current implementation of POP-C++ does not support object migration yet.

CHAPTER

3 User Manual

3.1 Introduction

The POP model (see chapter 2) is a suitable programming model for large heterogenous distributed environments but it should also remain as close as possible to traditional object oriented programming. Parallel objects of the POP model generalize sequential objects, keep the properties of object oriented programming (data encapsulation, inheritance and polymorphism) and add new properties.

The POP-C++ language is an extension of C++ that implements the POP model. Its syntax remains as close as possible to standard C++ so that C++ programmers can easily learn it and existing C++ libraries can be parallelized without much effort. Changing a sequential C++ application into a distributed parallel application is rather straightforward.

Parallel objects are created using parallel classes. Any object that instantiates a parallel class is a parallel object and can be executed remotely. To help the POP-C++ runtime to choose a remote machine to execute the remote object, programmers can add object description information to each constructor of the parallel object. In order to create parallel execution, POP-C++ offers new semantics for method invocations. These new semantics are indicated thanks to five new keywords. Synchronizations between concurrent calls are sometimes necessary, as well as event handling; the standard POP-C++ library supplies some tools for that purpose. This chapter describes the syntax of the POP-C++ programming language and presents main tools available in the POP-C++ standard library.

3.2 Parallel Objects

POP-C++ parallel objects are a generalization of sequential objects. Unless the term **sequential object** is explicitly specified, a parallel object is simply referred to as an object in teh rest of this chapter.

3.2.1 Parallel Class

Developing POP-C++ programs mainly consists of designing and implementing parallel classes. The declaration of a parallel class begins with the keyword `parclass` followed by the class name and the optional list of derived parallel classes separated by commas:

```
parclass ExampleClass {
    /* methods and attributes */
    ...
};
```

or

```
parclass ExampleClass: BaseClass1, BaseClass2 {
    /* methods and attributes */
    ...
};
```

As in the C++ language, multiple inheritance and polymorphism are supported in POP-C++. A parallel class can be a stand-alone class or it can be derived from other parallel classes. Some methods of a parallel class can be declared as overridable (virtual methods).

Parallel classes are very similar to standard C++ classes. Nevertheless, same restrictions applied to parallel classes.

- All data attributes are protected or private;
- The objects do not access any global variable;
- There are no programmer-defined operators;
- There are no methods that return memory address references.

These restrictions are not a major issue in object-oriented programming and in some cases they can improve the legibility and the clearness of programs. The restrictions can be mostly worked around by adding `get()` and `set()` methods to access data attributes and by encapsulating global data and shared memory address variables in other parallel objects.

3.2.2 Creation and Destruction

The object creation process consists of several steps: locating a resource satisfying the object description (resource discovery), transmitting and executing the object code, establishing the communication, transmitting the constructor arguments and finally invoking the corresponding object constructor. Failures on the object creation will raise an exception to the caller. Section 3.4.2 will describe the POP-C++ exception mechanism.

As a parallel object can be accessible concurrently from multiple distributed locations (shared object), destroying a parallel object should be carried out only if there is no other reference to the object. POP-C++ manages parallel objects' life time by an internal reference counter. A null counter value will cause the object to be physically destroyed.

Syntactically, the creation and the destruction of a parallel object are identical to those of C++. A parallel object can be implicitly created by just declaring a variable of the type of parallel object on stack or using the standard C++ `new` operator. When the execution goes out of the current stack or the `delete` operator is used, the reference counter of the corresponding object is decreased.

3.2.3 Parallel Class Methods

Like sequential classes, parallel classes contain methods and attributes. Method can be public, or private while attribute must be either protected or private. For each method, the programmer should define the invocation semantics. These semantics, described in section 2.4, are specified by two keywords, one for each side:

- Interface side:
  - `sync`: Synchronous invocation. This is the default value. For example:
    `sync void method1();`
  - `async`: Asynchronous invocation. For example:
    `async void method2();`
- Object side:
  - `seq`: Sequential invocation. This is the default value. For example:
    `seq void method1();`
  - `mutex`: Mutex invocation. For example:
    `mutex int method2();`
  - `conc`: Concurrent invocation. For example:
    `conc float method3();`

The combination of the interface and the object-side semantics defines the overall semantics of a method. For instance, the following declaration defines an synchronous concurrent method that returns an integer number:

```
sync conc int myMethod();
```

Figure 3.1 contains an example of a method `sort()` that has two arguments: an array of integer data (for input and output) and its (integer) size.

3.2.4 Object Description

Object descriptions are used to describe the resource requirements for the execution of the object. Object descriptions are declared along with parallel object constructor statements. Each constructor of a parallel object can be associated with an object description that resides directly after the argument declaration. The syntax of an object descriptor is as follows::

Figure 3.1 Array argument example

```
parclass Table {
    ...
    void sort([in, out, size=n] int *data, int n);
    ...
};

    /* main program */
    ...
    Table sales;
    int amount[10];
    sales.sort(amount, 10);
    ...
```

`@{`$expressions$`}`

An object description contains a set of resource requirement expressions. All resource requirement expressions are separated by semicolons and can be any of the following:

`od.`res_N`(`$exact$`);`

`od.`res_N`(`$exact$`, `$lbound$`);`

`od.`res_S`(`$resource$`);`

res_N := `power` | `memory` | `network` | `walltime`

res_S := `protocol` | `encoding` | `url`

Both $exact$ and $lbound$ terms are numeric expressions, and $resource$ is a null-terminated string expression. The semantics of those expressions depend on the resource requirement specifier (the keyword corresponding to res_N or res_S). The $lbound$ term is only used in non-strict object descriptions, to specify the lower bound of the acceptable resource requirements.

The current implementation allows indicating resources requirement in terms of:

- Computing power (in Mflops), keyword `power`
- Memory size (in MB), keyword `memory`
- Bandwidth (in Mb/s), keyword `network`
- Location (host name or IP address), keyword `url`
- Protocol (`"socket"` or `"http"`), keyword `protocol`
- Data encoding (`"raw"`, `"xdr"`, `"raw-zlib"` or `"xdr-zlib"`), keyword `encoding`

An example of object description is given in the figure 3.2. There, the constructor for the parallel object `Bird` requires the computing power of `P` Mflops, the desired memory space of 100MB (having 60MB is acceptable) and the communication protocol is socket or HTTP (socket has higher priority).

Object descriptors are used by the POP-C++ runtime system to find a suitable resource for the parallel object. Matching between object descriptors and resources is carried out by a multi-layer filtering technique: first, each expression (item) in every object descriptor will be evaluated and

Figure 3.2 Object descriptor example

```
parclass Bird
{
public:
    Bird(float P) @{ od.power(P);
                     od.memory(100,60);
                     od.protocol("socket http"); };
    ...
};
```

categorized (e.g., power, network, memory). Then, the matching process consists of several layers; each layer filters single category within object descriptors and performs matching on that category. Finally, if an object descriptor pass all filters, the object is assigned to that resource.

If no suitable resource is found to execute the objet then an exception is raised (see section 3.4.2).

3.2.5 Data marshaling

When calling remote methods, the arguments must be transferred to the object being called (the same happens for returned values). In order to operate with different memory spaces and different architectures, data is marshaled into a standard format prior to be send to remote objects. All data passed is serialized (marshalled) at the caller side and deserialized (demarshaled) at the callee side.

Programmers can help the POP-C++ compiler to generate efficient code by optionally specifying which arguments to transfer. This is done using an argument information block that can contain the directives `in` (for input), `out` (for output), or both. The argument information block should appear between braces (`[` and `]`), right before each argument declaration. Only input arguments are transferred from the caller to the remote object. Output arguments will only be transferred back to the caller for a synchronous method invocation. Without those directives, in the current implementation of POP-C++ the following rules are applied:

- If the method is asynchronous, arguments are input-only.
- If the method is synchronous:
  - Constant and passing-by-value arguments are input-only.
  - Other arguments are considered as both input and output.

POP-C++ automatically marshal/demarshal all the basic types of C++ (`int, float, char,` ... etc.). For arrays arguments programmers have to explicitly supply the number of elements the array contains. This is done using the directive `size` in the argument information block.

Void pointers (`void*`) cannot be used as arguments of parallel object methods.

For structured data, the programmer must supply a marshalling and demarshalling function through the directive `proc=`<*function name*> in the argument information block (see subsection 3.2.7).

Finally to pass sequential objects as arguments to a `parclass` method, programmers must derive their classes from a POP-C++ system class called `POPBase` and implement the virtual method `Serialize` (see subsection 3.2.6).

The POP-C++ system library provides two classes to support user specific marshalling/demarshalling functions: POPBuffer representing a system buffer that store marshalled data and POPMeemSpool representing the temporary memory spool that can be used to allocate temporary memory space for method invocation. The interfaces of these two classes are discussed bellow:

```
class  POPBuffer
{
public:
   void Pack(const Type *data, int n);
   void UnPack(Type *data, int n);
};

class POPMemSpool
{
public:
   void *Alloc(int size);
};
```

The POPBuffer class contains a set of Pack/UnPack methods for all simple data types Type (char,bool,int,float, etc.). Pack is used to marshal the array of data of size n. Pack is used to demarshal the data from the received buffer.

3.2.6 Marshalling Sequential Objects

To be able to pass sequential objects as arguments to a parclass method, programmers must derive their classes from a POP-C++ system class called POPBase and implement the virtual method Serialize. The interface of POPBase is described as following.

```
class POPBase
{
 public:
  virtual void Serialize(POPBuffer &buf, bool pack);
};
```

The method Serialize requires two arguments: the buf that stores the object data and flag pack specifying if it is to serialize data into the buffer or to deserialize data from the buffer.

Figure 3.3 shows an example of marshalling Speed class compared to Fig.3.4. Instead of specifying the marshalling function, the programmer implements the method Serialize of the POPBase class.

3.2.7 Marshalling Data Structures

For marshalling/demarshalling complex structures which are not objects, such as struct of C++, programmers need to indicate which function to use for marshalling/demarshalling the structure. In addition it is necessary to allocate temporary memory to store the structure to be sent. This memory

Figure 3.3 Marshalling an object

```
class Speed: public POPBase {
   public:
     Speed();
     virtual void Serialize(POPBuffer &buf, bool pack);

     float *val;
     int count;
};
void Speed::Serialize(POPBuffer &buf, bool pack) {
  if (pack) {
    buf.Pack(&count,1);
    buf.Pack(val, count);
  }
  else {
    if (val!=NULL) delete [] val;
    buf.UnPack(&count,1);
    if (count>0) {
      val=new float[count];
      buf.UnPack(val, count);
    }
    else val=NULL;
  }
}

parclass Engine {
    ...
    void accelerate(const Speed &data);
    ...
};
```

space will be then freed automatically by the system after the invocation is finished. POP-C++ provides a class `POPMemSpool` with the method `Alloc` to do this temporary memory allocation as well as a way to indicate, when calling a method, the function to use for marshalling/demarshalling an argument (`proc=`).

Figure 3.4 shows an example of data structure marshalling in POP-C++. In this example, the programmer provides the function `marsh()` for marshalling/demarshalling the argument `data` of method `accelerate()` of parallel class `Engine`. The programmer provided marshalling function `marsh()` takes five arguments:

- `buffer`: a buffer to marshal data into or demarshal from.
- `data`: the data structure to be marshalled or demarshalled, passed by reference.
- `count`: the number of elements to marshal or demarshal.
- `flag`: a bit mask that specifies where this function is called (marshalling or demarshalling, interface side or server-side).

Figure 3.4 Marshalling a structure

```
struct Speed {
    float *val;
    int count;
};

void marsh(POPBuffer &buffer, Speed &data, int count,
                        int flag, POPMemSpool *tmpmem) {
  if (flag & FLAG_MARSHAL)
  {
    buffer.Pack(&data.count,1);
    buffer.Pack(data.val, data.count);
  }
  else
  {
    buffer.UnPack(&data.count,1);
    //performing temporary allocation before calling UnPack
    data.val=(float *)tmpmem->Alloc(data.count*sizeof(float));
    buffer.UnPack(data.val, data.count);
  }
}

parclass Engine {
    ...
    void accelerate([proc=marsh] const Speed &data);
    ...
};
```

- `tmpmem`: a temporary memory spool (POPMemspool).

The marshalling function should be implemented in such a way that, when called to marshal, it packs all relevant fields of `data` into `buffer`. Likewise, when called to unmarshall, it should unpack all `data` fields from `buffer`. The `buffer` has two methods, overloaded for all scalar C++ types, to be used to pack and unpack data. These methods are `Pack()` and `UnPack()` respectively. Both methods are used with the number of items to pack, one for scalars, more than one for vectors.

`data` is passed to the marshalling function by reference, so the function can modify it if necessary. Also, if `data` is a vector (not the case shown in the example), the argument `count` will be greater than one.

A bit mask (`flag`) is passed to the marshalling function to specify whether it should marshal or demarshal data. The bit mask contains several bit fields, and if the bit `FLAG_MARSHAL` is set, the function should marshal data. Otherwise, it should demarshal data. If the bit `FLAG_INPUT` is set, the function is called at the interface side. Otherwise, it is called at the object-server side.

The last argument of the function (`tmpmem`) should be only used to allocate temporary memory space. In the example, the Speed structure contains an array `val` of `count` elements. At the object-side, before unpacking `val`, we need to perform temporary memory allocation using the memory spool interface provided by `tmpmem`.

3.3 Object Layout

A POP-C++ application is build using several executable files. One of them is the main program file, used to start running the application. Other executable files contain the implementations of the parallel classes for a specific platform. An executable file can store the implementation of one or several parallel objects. Programmers can help the POP-C++ compiler to group parallel objects into a single executable file by using the directive `@pack()`.

Figure 3.5 Packing objects into an executable file

```
Stack::Stack(...) {
    ...
}
Stack::push(...) {
    ...
}
Stack::pop(...) {
    ...
}

@pack(Stack, Queue, List)
```

All POP-C++ objects to be packed in a single executable file should be included as arguments of the `@pack()` directive. It is required that among the source codes passed to the compiler, exactly one source code must contain `@pack()` directive. Figure 3.5 shows an example with a file containing the source code of a certain class `Stack`, and a `@pack()` directive requiring that in the same executable file should be packed the executable code for the classes `Stack`, `Queue` and `List`.

3.4 Class Library

Alongside with the compiler, POP-C++ supplies a class library. This library basically offers classes for dealing with synchronizations and exceptions. These library classes are described in this section.

3.4.1 Synchronization

POP-C++ provides several method invocation semantics to control the level of concurrency of data access inside each parallel object. Communication between threads using shared attributes is straightforward because all threads on the same object share the same memory address space.

When concurrent invocations happen, it is possible that they concurrently access an attribute, leading to errors. The programmer should verify and synchronize data accesses manually. To deal with this situation, it could be necessary to synchronize the concurrent threads of execution.

Figure 3.6 The POPSynchronizer class

```
class POPSynchronizer {
public:
    POPSyncronizer();
    lock();
    unlock();
    raise();
    wait();
};
```

The **synchronizer** is an object used for general thread synchronization inside a parallel object. Every synchronizer has an associated lock (as in a door lock), and a condition. Locks and conditions can be used independently of each other or not. The synchronizer class is presented in the figure 3.6.

Calls to `lock()` close the lock and calls to `unlock()` open the lock. A call to `lock()` returns immediately if the lock is not closed by any other threads. Otherwise, it will pause the execution of the calling thread until other threads release the lock. Calls to `unlock()` will reactivate one (and just one) eventually paused call to `lock()`). The reactivated thread will then succeed closing the lock and the call to `lock()` will finally return. Threads that must not run concurrently can exclude each other's execution using synchronizer locks. When creating a synchronizer, by default the lock is open. A special constructor is provided to create it with the lock already closed.

Figure 3.7 Using the synchronizer lock

```
parclass Example {
private:
     POPSynchronizer syn;
     int counter;
public:
    int getNext() {
        syn.lock();
        int r = ++ counter;
        syn.unlock;
        return r;
    }
};
```

Conditions can be waited and raised. Calls to `wait()` cause the calling thread to pause its execution until another thread triggers the signal by calling `raise()`. If the waiting thread possess the lock, it will automatically release the lock before waiting for the signal. When the signal occurs, the waiting thread will try to re-acquire the lock that it has previously released before returning control to the caller.

Many threads can wait for the same condition. When a thread calls the method `raise()`, all waiting-for-signal threads are reactivated at once. If the lock was closed when the `wait()` was called, the reactivated thread will close the lock again before returning from the `wait()` call. If other threads calls `wait()` with the lock closed, all will wait the lock to be open again before they are actually reactivated.

The typical use of the synchronizer lock is when many threads can modify a certain property at the same time. If this modification must be done atomically, no other thread can interfere before it is finished. The figure 3.7 shows an example of this synchronizer usage.

The typical use of a synchronizer condition is when some thread produces some information that must be used by another, or in a producer-consumer situation. Consumer threads must wait until the information is available. Producer threads must signal that the information is already available. Figure 3.8 is an example that shows the use of the condition.

Figure 3.8 Using the synchronizer condition

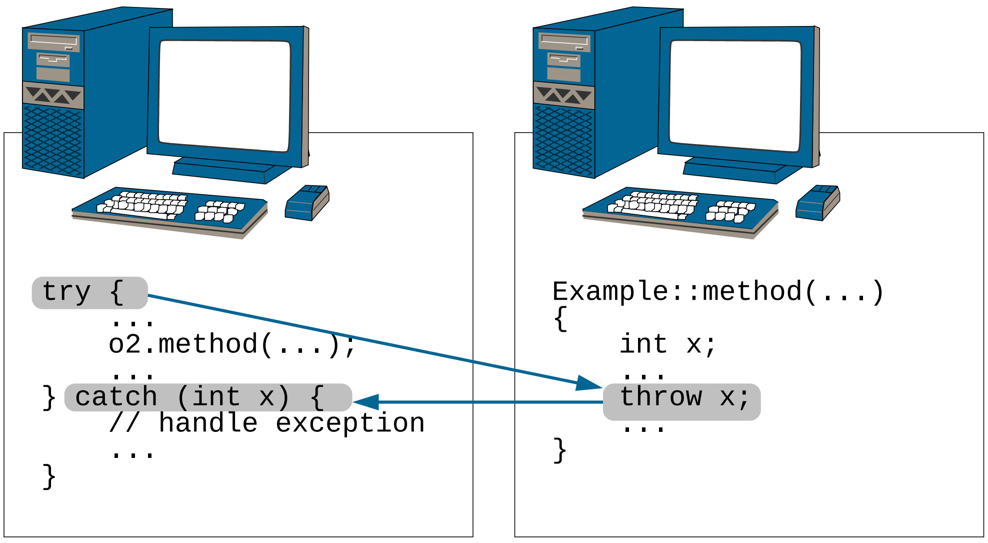
```
parclass ExampleBis {
private:
    int cakeCount;
    boolean proceed;
    Synchronizer syn;
public:
    void producer(int count) {
        cakeCount = count;
        syn.lock();
        proceed = true;
        syn.raise();
        syn.unlock();
    }
    void consumer() {
        syn.lock();
        if (!proceed) wait();
        syn.unlock();
        /* can use cakeCount from now on... */
    }
};
```

3.4.2 Exceptions

Errors can be efficiently handled using exceptions. Instead of handling each error separately based on an error code returned by a function call, exceptions allow the programmer to filter and centrally manage errors trough several calling stacks. When an error is detected inside a certain method call, the program can throw an exception that will be caught somewhere else.

The implementation of exceptions in non-distributed applications, where all components run within the same memory address space is fairly simple. The compiler just need to pass a pointer to the exception from the place where it is thrown to the place where it is caught. However, in distributed environments where each component is executed in a separate memory address space (and eventually data are represented differently due to heterogeneity), the propagation of exception back to a remote component is complex.

Figure 3.9 Exception handling example



```
try {
    ...
    o2.method(...);
    ...
} catch (int x) {
    // handle exception
    ...
}
```

```
Example::method(...)
{
    int x;
    ...
    throw x;
    ...
}
```

POP-C++ supports transparent exception propagation. Exceptions thrown in a parallel object will be automatically propagated back to the remote caller (figure 3.9). The current POP-C++ prototype allows the following types of exceptions:

- Scalar data (`int`, `float`, etc.)
- Parallel objects
- Objects of class `POPException` (system exception)

All other C++ types (`struct`, `class`, vectors) will be converted to `POPException` with the `UNKNOWN` exception code.

The invocation semantics of POP-C++ affect the propagation of exceptions. For the moment, only synchronous methods can propagate the exception. Asynchronous methods will not propagate any exception to the caller. POP-C++ current behavior is to abort the application execution when such exception occurs.

Besides the exceptions created by programmers, POP-C++ uses an exception of type `POPException` to notify the user about the following system failure:

- Parallel object creation fails. It can happen due to the unavailability of suitable resources, an internal error on POP-C++ services, or the failures on executing the corresponding object code.
- Parallel object method invocation fails. This can be due to the network failure, the remote resource down, or other causes.

The interface of `POPException` is described bellow:

```
class POPException
{
public:
   const paroc_string Extra()const;
   int Code()const;
   void Print()const;
};
```

`Code()` method returns the corresponding error code of the exception. `Extra()` method returns the extra information about the place where the exception occurs. This extra information can contains the parallel object name and the machine name where the object lives. `Print()` method prints atext describing the exception.

All exceptions that are parallel objects are propagated by reference. Only the interface of the exception is sent back to the caller. Other exceptions are transmitted to the caller by value.

3.5 Coupling MPI code

POP-C++ can encapsulate MPI processes in parallel objects, allowing POP-C++ applications to use existing HPC MPI libraries. Each MPI process will become a parallel object in POP-C++. The user can control the MPI-based using:

- Standard POP-C++ remote method invocations. This allows the user to initialize data or computation on some or all MPI processes.
- MPI communication primitives such as `MPI_Send`, `MPI_Recv`, etc. These primitives will use vendor specific communication protocol (e.g. Myrinet/GM).

Each MPI process in POP-C++ will become a parallel object of identical type that can be accessed from outside through remote method invocations.

Figure 3.10 shows an example of using MPI in POP-C++. `TestMPI` methods contains some MPI code. Users need to implement a method named `ExecuteMPI`. This method is invoked on all MPI processes. In this case, the method will broadcast the local value `val` of process 0 to all other processes.

Since an MPI program requires special treatment at startup (mpirun, MPI_Initialize, etc.), users must use a POP-C++ built-in class template `POPMPI` to create parallel object-based MPI processes. Figure 3.11 illustrates how to start and to call MPI processes. We first create 2 MPI processes of type `TestMPI` using the template class `POPMPI` (variable `mpi`). Then we can invoke methods on a specific MPI process using its rank as the index. `ExecuteMPI` is a pre-defined method of `POPMPI` which will then invokes all corresponding `ExecuteMPI` methods of the MPI parallel objects (`TestMPI`).

The declaration of `POPMPI` is described as follows:

```
template<class T> class POPMPI
{
```

Figure 3.10 MPI parallel objects

```
parclass TestMPI {
public:
  TestMPI();
  async void ExecuteMPI();
  async void Set(int v);
  sync void Get();
private:
   int val;
};

TestMPI::TestMPI() {
  val=0;
}

void TestMPI::ExecuteMPI() {
  MPI_Bcast(&val,1,MPI_INT, 0, MPI_COMM_WORLD);
}

void TestMPI::Set(int v) {
  val=v;
}
int TestMPI::Get() {
  return val;
}



 public:
  POPMPI(); //Do not create MPI process
  POPMPI(int np); // Create np MPI process of type T
  ~POPMPI();

  bool Create(int np); // Create np MPI process of type T
  bool Success(); // Return true if MPI is started.
                  // Otherwise, return false
  int GetNP(); //Get number of MPI processes

  bool ExecuteMPI(); //Execute method ExecuteMPI on all processes
  inline operator T*(); //type-cast to an array of parclass T
};
```

Figure 3.11 Creating MPI parallel objects

```
#include <popc_mpi.h>
int main(int argc, char **argv) {
  POPMPI<TestMPI> mpi(2);
  mpi[0].Set(100); //Set on MPI process 0
  printf(‘‘Values before: proc0=%d, proc1=%d\n’’,
                       mpi[0].Get(), mpi[1].Get());
  mpi.ExecuteMPI(); //Call ExecuteMPI methods on all MPI processes
  printf(‘‘Values after: proc0=%d, proc1=%d\n’’,
                       mpi[0].Get(), mpi[1].Get());
}

----------
Output of the program:
Values before: proc0=100, proc1=0
Values after: proc0=100, proc1=100
```

3.6 Limitations

There are certain limitations to the current implementation of POP-C++. Some of these restrictions are expected to disappear in the future while others are simply due to the nature of parallel programming and the impossibility for parallel objects to share a common memory. For the current version (1.3), the limitations are:

- A parallel class cannot contain public attributes.
- A parallel class cannot contain a class attribute (`static`).
- A parallel class cannot be template.
- An asynchronous method cannot return a value and cannot have output parameters.
- Global variables exist only in the scope of parallel objects (`@pack()` scope).
- The programmer must specify the size of pointer parameters in remote method invocation as they are considered as arrays.
- A parallel object method cannot return a memory address.
- Sequential classes used as parameter must be derived from `POPBase` and the programmer must implement the `Serialize` method.
- Parameters must have exactly the same dynamic type as in method declaration, an object of a derived class cannot be used (polymorphism).
- Exceptions. Only scalar, parallel object and POPException type are handled. All other exceptions are converted to POPException with the *unknown* code.
- Exceptions raised in an asynchronous method are not propagated. They abort (cleanly) the application.

CHAPTER

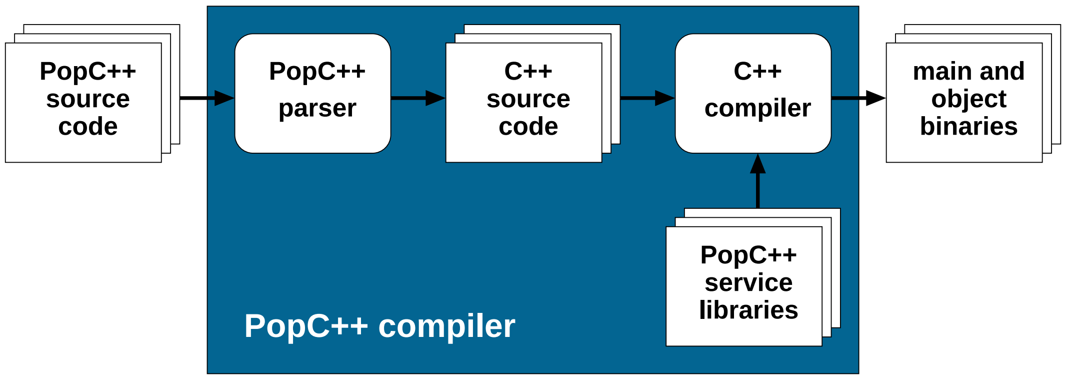
4 Compiling and Running

POP C++

4.1 Compilation

The POP-C++ compiler generates a main executable and several object executables. The main executable provides a starting point to the application and object executables are loaded and started by the POP-C++ runtime system whenever a parallel object is created. The compilation process is illustrated in figure 4.1.

Figure 4.1 POP-C++ compilation process



The POP-C++ compiler contains a parser which translates the code to ANSI C++ code. Service libraries provide APIs that manages communication, resource discovery and object allocation, etc. An ANSI C++ compiler finally generates binary executables.

4.2 Example Program

We will see in this section how to write a simple POP-C++ program.

4.2.1 Programming

integer.ph

Figure 4.2 shows the declaration of a parallel class in a POP-C++ header. From the language aspect, this part contains the major differences between POP-C++ and C++. However, as the example shows, a POP-C++ class declaration is similar to a C++ class declaration with the addition of some new keywords. A parallel class consists of constructors (lines 3 and 4), destructor (optional), interfacing methods (`public`, lines 5-7), and a data attribute (`private`, line 9).

Figure 4.2 File `integer.ph`

```
 1: parclass Integer {
 2: public :
 3:     Integer(int wanted, int minp) @{ od.power(wanted, minp); };
 4:     Integer(POPString machine) @{ od.url(machine);};
 5:     seq async void Set(int val);
 6:     conc int Get();
 7:     mutex void Add(Integer &other);
 8: private :
 9:     int data;
10: };
```

In the figure 4.2, the programmer defines a parallel class called `Integer` starting with the keyword `parclass` (line 1). Two constructors (lines 3 and 4) of `Integer` are both associated with two object descriptors which reside right after the argument declaration, between `@{...}`. The first object descriptor (line 3) specifies the parameterized high level requirement of resource (i.e. computing power). The second object descriptor (line 4) is the low-level description of the location of resource on which the object will be allocated.

The invocation semantics are defined in the class declaration by putting corresponding keywords (`sync`, `async`, `mutex`, `seq`, `conc`) in front of the method declaration. In the example of figure 4.2, the `Set()` method (line 5) is sequential asynchronous, the `Get()` method (line 6) is concurrent and the `Add()` method (line 7) is mutual exclusive execution. Although it is not shown in the example the user can also use standard C++ features such as `virtual`, `const`, or inheritance with the parallel class.

integer.cc

The implementation of the parallel class `Integer` is shown in figure 4.3. This implementation does not contain any invocation semantics and looks similar to a C++ code, except at line 18 where we provide a directive `@pack` to tell the POP-C++ compiler the place to generate the parallel object executable for `Integer` (see section 3.3 for the `pack` directive).

main.cc

The main POP-C++ program in figure 4.4 looks exactly like a C++ program. Two parallel objects of type `Integer`, `o1` and `o2`, are created (line 6). The object `o1` requires a resource with the

Figure 4.3 File `integer.cc`

```
#include "integer.ph"

Integer::Integer(int wanted, int minp) {}
Integer::Integer(POPString machine) {}

void Integer::Set(int val) {
    data = val;
}

int Integer::Get() {
    return data;
}

void Integer::Add(Integer &other) {
    data = other.Get();
}

@pack(Integer);
```

desired performance of 100MFlops although the minimum acceptable performance is 80MFlops. The object `o2` will explicitly specify the resource location (localhost).

After the object creations, the invocations to methods `Set()` and `Add()` are performed (line 7-9). The invocation of `Add()` method shows an interesting property of the parallel object: the object `o2` can be passed from the `main` program to the remote method `Add()` of parallel object `o1`.

Lines 12-15 illustrate how to handle exceptions in POP-C++ using the keyword pair `try` and `catch`. Although `o1` and `o2` are distributed objects but the way to handle the remote exceptions is the same as in C++.

4.2.2 Compiling

We generate two executables: the main program (`main`) and the object code (`integer.obj`). POP-C++ provides the command `popcc` to compile POP-C++ source code. To compile the main program we use the following command:

```
popcc -o main integer.ph integer.cc main.cc
```

4.2.3 Compile the object code

Use `popcc` with option `-object` to generate the object code:

```
popcc -object -o integer.obj integer.ph integer.cc
```

Figure 4.4 File `main.cc`

```
#include "integer.ph"

int main(int argc, char **argv)
{
    try {
        Integer o1(100, 80), o2("localhost");
        o1.Set(1);
        o2.Set(2);
        o1.Add(o2);
        printf("Value=%d\n", o1.Get());
    }
    catch (POPException *e) {
        printf("Object creation failure\n");
        e->Print();
        return -1;
    }
    return 0;
}
```

You can note that we have compiled the declaration of the parallel class `integer.ph` explicitly. The user can also generate intermediate code `.o` that can be linked using a C++ compiler by using the `-c` option (compile only) with `popcc`.

Compilation for several parclasses with dependencies

The compilation is a little bit more difficult for more complex applications using several different parallel classes. This is the case, for example, when the main program calls methods from objects of different parallel classes or when there is a chain of dependencies between the main program and several parallel classes as illustrated on figure 4.5.

Figure 4.5 Parclasses with dependencies

$$\begin{tikzcd}
\boxed{\text{main}} \arrow[r] & \boxed{\text{myobj1}} \arrow[r] & \boxed{\text{myobj2}} \arrow[r] & \boxed{\text{myobj3}} \arrow[r] & \boxed{\text{myobj4}}
\end{tikzcd}$$

Since each class contains some internal POP-C++ classes such as the `interface` or the `broker` classes, the compilation must avoid to create multiple definitions of these classes. An easy way to avoid this is to begin the compilation with the last class of the chain (the class `myobj4` on figure 4.5) and then to compile each parallel class in reverse order. To compile any class in the chain we needs the parallel classe which is directly after the one we are compiling in the chain of dependency. When compiling a parallel classe without generating the executable code (option `-c`), the POP-C++ compiler generates a relocatable object file called className`.stub.o`. In addition the POP-C++ compiler has an option called `-parclass-nobroker` which allows to generate

Figure 4.6 How to compile applications with dependencies

```
popcc -object -o myobj4.obj myobj4.ph myobj4.cc
popcc -c -parclass-nobroker myobj4.ph

popcc -object -o myobj3.obj myobj3.ph myobj3.cc myobj4.stub.o
popcc -c -parclass-nobroker myobj3.ph

popcc -object -o myobj2.obj myobj2.ph myobj2.cc myobj3.stub.o
popcc -c -parclass-nobroker myobj2.ph

popcc -object -o myobj1.obj myobj1.ph myobj1.cc myobj2.stub.o

popcc -o main main.cc myobj1.ph myobj1.cc myobj2.stub.o
```

relocatable code without internal POP-C++ classes. The way to compile a POP-C++ application with dependencies as illustrated on figure 4.5 is shown on figure 4.6.

The source code of this example can be found in the `examples` directory of the POP-C++ distribution and on the POP-C++ web site (`http://gridgroup.hefr.ch/popc`).

4.2.4 Running

To execute a POP-C++ application we need to generate the object map file which contains the list of all parallel classes used by the application. For each parallel class we need to indicate for which architecture the compilation has been done and the location of the file (object file).

With POP-C++ it is possible to get this information by executiong the object file with the option `-listlong`.

Example for the `Interger` parallel class:

```
    ./integer.obj -listlong
```

To generate the object map file we simply redirect the output to the object map file:

```
    ./integer.obj -listlong > obj.map
```

The object map file contains all mappings between object name, platform and the executable location. In our example we have compiled on Linux machines and the object map file looks like this:

```
    Integer i686-pc-Linux /home/myuser/popc/test/integer/integer.obj
```

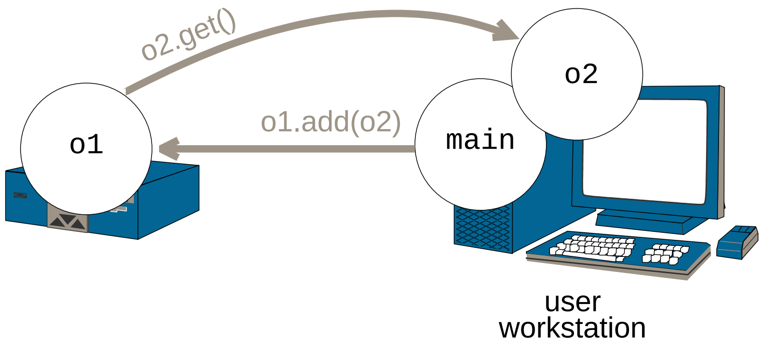
If you also compile the object code for another platform (e.g. Solaris), simply add a similar line to `obj.map`. The executable location can also be an absolute path or an URL (HTTP or FTP).

We can now run the program using the command `popcrun`:

```
    popcrun obj.map ./main
```

Figure 4.7 shows the execution of `Integer::Add()` method on line 4 in figure 4.3 of the example. The system consists of three running processes: the `main`, object `o1` and object `o2`. The `main` is started by the user. Objects `o1` and `o2` are created by `main`. Object `o2` and the `main` program run on the same machine although they are in two separate memory address spaces; object `o1` runs on a remote machine. The `main` invokes the `o1.Add()` with the interface `o2` as an argument. Object `o1` will then connect to `o2` automatically and invoke the method `o2.Get()` to get the value and to add this value to its local attribute `data`. POP-C++ system manages all object interactions in a transparent manner to the user.

Figure 4.7 An execution example



CHAPTER

5 Installation Instructions

5.1 Before installing

To find out about the latest sources releases or installation instructions please visite our wiki :

`http://gridgroup.hefr.ch/popc`

POP-C++ is built on top of several widely known software packages. The following packages of are required before compiling.

- a C++ compiler (g++)
- zlib-devel
- the Gnu Bison (optional)
- the Globus Toolkit (optional)

Before installation we should make the following configuration choices. In case of doubt the default values can be used.

- The compilation directory that should hold roughly 50MB. This directory will contain the distribution tree and the source files of POP-C++. It may be erased after installation.
- The installation directory that will hold less than 40MB. It will contain the compiled files for POP-C++, include and configuration files. This directory is necessary in every computer executing POP-C++ programs. (by default `/usr/local/popc`)
- A temporary directory will be asked in the installation process. This directory will be used by POP-C++ to hold files during the applications execution. (by default `/tmp`)
- Resource topology. The administrator must choose what computers form our grid.

5.2 Standard Installation

This section explains how to install POP-C++ using all default options. This is usually sufficient if you want to test POP-C++ on your desktop computer. An advanced installation is explained in the two sections below.

POP-C++ distribution uses standard GNU tools to compile. The following commands will generate all necessary files:

```
cd compilation-directory
tar xzf popc-version.tar.gz
cd popc-version
./configure
make
```

The `make` command takes a while to finish. After it, all files will be compiled and POP-C++ is ready for installation. To install POP-C++ type:

```
make install
```

After copying the necessary files to the chosen installation directory, a setup script is run. It asks different questions, and the information gathered before the installation should be sufficient to answer it.

For a standard installation of POP-C++ it is sufficient to pick the simple installation when asked.

In case it is necessary to restart the setup script, it can be done with the following command:

```
installation-directory/sbin/popc_setup
```

5.3 Custom Installation

The configuration utility can be started with command line arguments for a custom installation. These arguments control whether some extra or different features should be enabled. A list of optional features can be found in the figure 5.1. The full list of options accepted by the configuration utility can be obtained with the `--help` argument.

Figure 5.1 Optional configuration features

`--enable-mpi`	Enable MPI-support in POP-C++
`--enable-globus=flavor`	Enable Globus-support in POP-C++
`--enable-xml`	Enable XML encoding in POP-C++
`--enable-http`	Enable HTTP communication protocol in POP-C++

The current distribution of POP-C++ 1.3 supports the following features:

- Globus-enabled services. POP-C++ allows to build the runtime services for Globus. We only use the Pre WS GRAM of Globus Toolkit (GT3.2 or GT4). To enable this feature, you will need to provide the Globus's built flavor (refer Globus documentation for more information). Before configuring POP-C++ with Globus, you need to set the environment variable GLOBUS_LOCATION to the Globus installed directory. Bellow is an example of configuring POP-C++ for Globus with the flavor of `gcc32dbgpthr`:

./configure --enable-globus=gcc32dbgpthr

- Enable SOAP/XML encoding. POP-C++ supports multiple data encoding methods as the local plugins to the applications. POP-C++ requires the Xerces-C library to enable SOAP/XML encoding (configure –enable-xml).
- Enable HTTP protocol in parallel object method invocations. This protocol allows objects to communicate cross sites over the firewall (experimental feature).
- Enable MPI support. This feature allows POP-C++ applications to implement parallel objects as MPI processes (refer section 3.5).

5.4 Configuring POP-C++ services

The POP-C++ runtime service is a fully distributed model where resource connectivity is represented as a dynamic graph. A resource (POP-C++ service node) can join the environment by registering itself to a node (or a master) inside this environment (dynamic) or by being listed statically in the “known nodes” of other resources inside the environment.

When configuring POP-C++ services on each node, the user will be prompted to give information about the master nodes (to which the configuring POP-C++ service will register itself to) and about the child nodes that the configuring POP-C++ service will manage.

- The number of processors available on the resource (node). If the POP-C++ service represents a front end of a cluster, the number of processors is the number of nodes of that cluster. In this case, you will need to specify the script to submit a job to the cluster.
- The local username to be used in the child nodes, in the case POP-C++ is started by the `root` user (this item is optional).
- The TCP/IP port to be used by POP-C++ (this item is optional, by default the port 2711 is used).
- The domain name of the local resource (optional). If no domain is provided, the IP address will be used.

  Please note that more option can be set in appendix B : Runtime environment variables.

When you run “make install” for the first time, it will automatically execute the following script:

installation-directory`/sbin/popc_setup`

This script will ask you several question about the local resource and the execution environment.

We assume to configure POP-C++ on 25 workstations sb01.eif.ch-sb025.eif.ch. We choose the machine sb02.eif.ch as the master node and the rest will register to this machine upon starting the POP-C++ services. We configured POP-C++ with Globus Toolkit 4.0. The POP-C++installation is shared on NFS among all machines. Following is a transcript:

1. Configure POP-C++ service on your local machine:

POP-C++ runtime environment assumes the resource topology is a graph. Each node can join the environment by register itself to other nodes (master hosts). If you want to deploy POP-C++ services at your site, you can select one or several machines to be master nodes and when configure other nodes, you will need to enter these master nodes as requested. Another possibility to create your resource graph is to explicitly specify list of child nodes to which job requests can be forwarded to. Here is an example:

```
------------------
Enter the full qualified master host name (POPC gateway):
sb02.eif.ch

Enter the full qualified master host name (POPC gateway):
[Enter]

Enter the child node:
[Enter]
------------------
```

2. Information of the local execution environment:

   - Number of processors of the local machine. If you intend to run POP-C++ service on a front end of a cluster, this can be the number of nodes inside that cluster.
   - Maximum number of jobs that can be submitted to your local machine.
   - The local user account you would like to run jobs. This is only applied to the standalone POP-C++ services. In the case you use Globus to submit jobs, authentication and authorization are provided by Globus, hence, this information will be ignored.
   - Environment variables: you can set up your environment variables for your jobs. Normally, you need to set the **LD_LIBRARY_PATH** to all locations where dynamic libraries are found.

3. If you enable Globus while configuring POP-C++, information about Globus environment will be prompted:

   - The Globus gatekeeper contact: this is the host certificate of the local machine. If you intend to share the same Globus host certificate among all machines of your site, you should provide this certificate here instead of the Globus's gatekeeper contact.
   - Globus grid-mapfile: POP-C++ will need information from the Globus's grid-mapfile to verify if the user is eligible for running jobs during resource discovery.

   Here is an example of what you will be asked:

```
----------------
Enter number of processors available (default:1):
[Enter]

Enter the maximum number of POP-C++ jobs that can run
concurrently(default: 1):
[Enter]
```

```
Which local user you want to use for running POPC jobs?
[Enter]

CONFIGURING THE RUNTIME ENVIRONMENT

Enter the script to submit jobs to the local system:
[Enter]


Communication pattern:
```

NOTE: Communication pattern is a text string defining the protocol priority on binding the interface to the object server. It can contain “*” (matching non or all) and “?” (matching any) wildcards.

For example: given communication pattern “`socket://160.98.* http://*`”:

- If the remote object access point is “`socket://128.178.87.180:32427 http://128.178.87.180:8080/MyObj`”, the protocol to be used will be “http”.
- If the remote object access point is “`socket://160.98.20.54:33478 http://160.98.20.54:8080/MyObj`”, the protocol to be used will be “socket”.

```
SETTING UP RUNTIME ENVIRONMENT VARIABLES

Enter variable name:
LD_LIBRARY_PATH

Enter variable value:
/usr/openwin/lib:/usr/lib:/opt/SUNWspro/lib

Enter variable name:
[Enter]

DO YOU WANT TO CONFIGURE POPC SERVICES FOR GLOBUS? (y/n)
y

Enter the local globus gatekeeper contact:
/O=EIF/OU=GridGroup/CN=host/eif.ch

Enter the GLOBUS grid-mapfile([/etc/grid-security/grid-mapfile]):
[Enter]

=====================================================
CONFIGURATION POP-C++ SERVICES COMPLETED!
=====================================================
---------------
```

4. Generate startup script: you will be asked to generate startup scripts for POP-C++ services. These scripts (SXXpopc*) will be stored in the sbin subdirectory of the POP-C++ installed directory.

- The local port where POP-C++ service is running. It is recommended to keep the default port (2711).
- The domain name of the local host. If your machine is not listed in the DNS, just leave this field empty.
- Temporary directory to store log information. If you leave this field empty, /tmp will be used.
- If you configure POP-C++ with Globus, the Globus installed directory will also been prompted.

Bellow is the example:

```
---------------
Do you want to generate the POPC++ startup scripts? (y/n)
y
=====================================================
CONFIGURING STARTUP SCRIPT FOR YOUR LOCAL MACHINE...
Enter the service port[2711]:
[Enter]

Enter the domain name:
eif.ch

Enter the temporary directory for intermediate results:
/tmp/popc

DO YOU WANT TO GENERATE THE GLOBUS-BASED POPC SCRIPT? (y/n)
y

Enter the globus installed directory (/usr/local/globus-4.0.0):
[Enter]

CONFIGURATION DONE!
---------------
```

If you want to change the POP-C++ configuration, you can manually run the configure script **popc_setup** located in the *<installed directory>/sbin*

5.5 System Setup and Startup

The installation tree provides a shell setup script. It sets paths to the POP-C++ binaries and library directories. The most straightforward solution is to include a reference to setup script in the users login shell setup file (like `.profile`, `.bashrc` or `.cshrc`). The setup scripts (respectively for C-shells and Bourne shells) are:

installation-directory`/etc/popc-user-env.csh` and
installation-directory`/etc/popc-user-env.sh`

Before executing any POP-C++ application, the runtime system (job manager) must be started. There is a script provided for that purpose, so every node must run the following command:

installation-directory`/sbin/SXXpopc start`

`SXXpopc` is a standard Unix daemon control script, with the traditional `start`, `stop` and `restart` options. There is a different version to be used with Globus, called `SXXpopc.globus`.

APPENDIX

A Command Line Syntax

A.1 POP-C++ Compiler command

```
popcc [POP-C++ options] [other C++ options] sources...
POP-C++ options:
   -cxxmain:            Use standard C++ main (ignore POP-C++
                        initialization).
   -paroc-static:       Link with standard POP-C++ libraries
                        statically.
   -paroc-nolib:        Avoid standard POP-C++ libraries from
                        linking.
   -parclass-nointerface: Do not generate POP-C++ interface codes
                          for parallel objects.
   -parclass-nobroker:    Do not generate POP-C++ broker codes
                          for parallel objects.

   -object[=type]:      Generate parallel object executable
                        (linking only)(type: std (default) or mpi)
   -popcpp:             POP-C++ parser
   -cpp=<preprocessor>: C++ preprocessor command
   -cxx=<compiler>:     C++ compiler
   -parocld=<linker>:   C++ linker (default: same as C++ compiler)
   -parocdir:           POP-C++ installed directory
   -noclean:            Do not clean temporary files
   -verbose:            Print out additional information

Environment variables change the default values used by POP-C++:
   POPC_LOCATION:  Directory where POP-C++ has been installed
   POPC_CXX:       The C++ compiler used to generate object code
   POPC_CPP:       The C++ preprocessor
   POPC_LD:        The C++ linker used to generate binary code
   POPC_PP:        The POP-C++  parser
```

APPENDIX

B Runtime environment variables

The following environment variables affect or change the default behaviors of the POP-C++ runtime. To ensure that the environment of all running objects these variables should all be set during the installation `make install` or in the environment setup script `popc-runtime-env`.

`POPC_LOCATION:`	Location of installed POP-C++ directory.
`POPC_PLUGIN_LOCATION:`	Location where additional communication and data encoding plugins can be found.
`POPC_JOBSERVICE:`	The access point of the POP-C++ job manager service. If the POP-C++ job manager does not run on the local machine where the user start the application, the user must explicitly specify this information. Default value: socket://localhost:2711.
`POPC_HOST:`	Full qualified host name of local node. This host name will be interpreted
`POPC_IP:`	IP of local node. Only used if POPC_HOST is not defined
`POPC_IFACE:`	If POPC_HOST and POPC_IP are not set, use this interface to determine node IP. If not set, the default gateway interface is used.
`POPC_PLATFORM:`	The platform name of the local host. By default, the following format is used: <cpu id>-<os vendor>-<os name>.
`POPC_MPIRUN:`	The mpirun command to start POP-C++ MPI objects.

```
POPC_JOB_EXEC:          Script used by the job manager to submit
                         a job to local system.

POPC_DEBUG:             Print all debug information.
```

Bibliography

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. *GridFTP Protocol Specification*. GGF GridFTP Working Group Document, September 2002. http://www.globus.org/research/papers.htm.

[2] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[3] I. Foster and N. Karonis. A grid-enabled mpi: Message passing in heterogeneous distributed computing systems. In *Proc. 1998 SC Conference*, November 1998.

[4] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[5] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.

[6] Andrew Grimshaw, Adam Ferrari, Fritz Knabe, and Marty Humphrey. Legion: An operating system for wide-area computing. *IEEE Computer*, 32:5:29–37, May 1999.

[7] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 2003.

[8] Kuonen P. Nguyen, T. A. Programming the grid with pop-c++. *Future Generation Computer Systems (FGCS)*, 23(1):23–30, January 2007.

[9] Tuan-Anh Nguyen. *An Object-oriented model for adaptive high performance computing on the computational Grid*. PhD thesis, Swiss Federal Institute of Technology-Lausanne, 2004.

[10] Object Management Group, Framingham, Massachusetts. *The Common Object Request Broker: Architecture and Specification — Version 2.6*, December 2001.

[11] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-service for message passing programs. In *Proc. of the IEEE/ACM SC2000 Conference*, November 2000.

[12] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC2001)*, 2001. San Francisco, California.

[13] Weiqin Tong, Jingbo Ding, and Lizhi Cai. A parallel programming environment on grid. In *International Conference on Computational Science 2003*, pages 225–234, 2003.

[14] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In IEEE Press, editor, *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, 2003.

[15] Tiffani L. Williams and Rebecca J. Parsons. The heterogeneous bulk synchronous parallel model. *Lecture Notes in Computer Science*, 1800, 2000.