# Parallel Object Programming C++
# User and Installation Manual


Version 1.1

Tuan Anh Nguyen
Marcelo Pasin
Pierre Kuonen

Grid and Ubiquitous Computing Group

University of Applied Sciences
of Western Switzerland, Fribourg

Parallel Object Programming C++
User and Installation Manual
Manual version: 1.1



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

# Contents

CHAPTER

# 1 Introduction and Background



## 1.1 Introduction

Grid computing has become in the last few years a dominant theme in the parallel and distributed computing domain. Although many researches have been focused on enabling grid infrastructures for scientific computing such as resource management and discovery [4, 6, 2], grid service architecture [5], grid security [13], grid data management [1, 11], etc., programming on a grid remains a hard task. Recent efforts to port traditional programming tools to the grid such as MPI [3, 10, 7] or BSP [12, 14], have had some success. These tools allow programmers to run their existing parallel applications on the grid. However, efficient exploitation of the grid performance regarding its heterogeneity still needs to be manually controlled and tuned by programmers.

A parallel object-oriented programming system, called POP-C++, was developed to allow programmers to exploit the heterogeneous performance of the grid for their applications (POP stands for Parallel Object Programming). POP-C++ system provides supports for grid programming and deployment supports at different levels: from a programming tool (POP-C++ language and compiler) for writing grid applications to the runtime services for running applications on the grid.

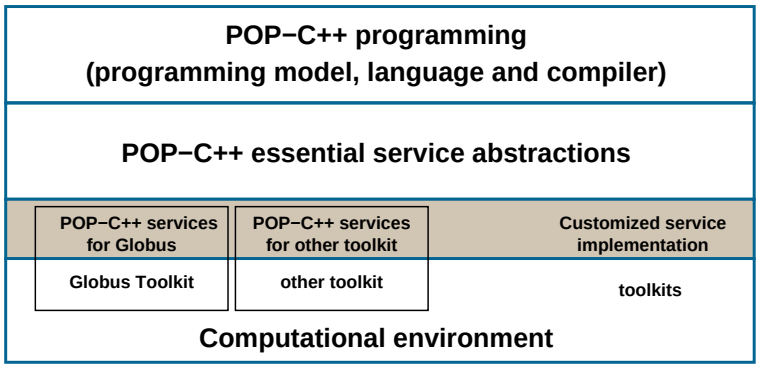
Inspired by CORBA [9] and C++, the POP-C++ programming language extends C++ by adding a new type of **parallel object**, allowing to run C++ objects on distributed environments. With POP-C++, programming on the grid is as simple as writing a C++ program. POP-C++ services are mainly used for interfacing POP-C++ applications and the grid environment.

## 1.2 System Overview

Although POP-C++ programming system focuses on an object-oriented programming model for the grid it also includes a runtime system which is responsible for providing necessary services for running POP-C++ applications on the grid. The overview of POP-C++ system is illustrated in figure 1.1.

The POP-C++ runtime system consists of three layers: the service layer the essential service abstractions layer, and the programming layer. The service layer is built to interface with lower level grid toolkits (e.g. Globus). The essential service abstractions layer provides an abstract interface for the programming layer. On top of the architecture is the programming layer, which provides

**Figure 1.1** POP-C++ structure




necessary support for developing grid-enabled object-oriented applications. More details of the POP-C++ runtime layers are given in a separate document [8].

## 1.3 Programming Model

POP-C++ programming model introduces a new type of object: the **parallel object**. Parallel objects coexist and cooperate with sequential objects during the application execution. Parallel objects in POP-C++ generalize the sequential object by keeping the advantages of object-orientation such as data encapsulation, inheritance and polymorphism and by adding new properties to the object:

- Distributed shareable objects
- Dynamic and transparent object allocation driven by the high-level requirement descriptions
- Various method invocation semantics

## 1.4 Structure of the Text

This manual is structured in five chapters, including this introduction. The second chapter is devoted to explain the programming model of POP-C++. The third chapter describes the programming syntax of POP-C++ extensions over C++. The fourth chapter explains how to compile and run applications written using the POP-C++ language. The fifth chapter shows how to compile and install POP-C++. Programmers interested on using POP-C++ should read chapters 2, 3 and 4. System managers should read chapter 5, and eventually chapters 2 ad 4.

CHAPTER

# 2 Parallel Object Model



## 2.1 Introduction

Object-oriented methods provide high level abstractions for software engineering. The nature of objects shows many possibilities of parallelism. There can be a collection of objects where each object may live independently from others. There is also parallelism inside each object: some operations on the same object can occur concurrently. In distributed environments such as the GRID, having all objects running remotely is usually not efficient due to the communication bottleneck problem. Thus, the two questions must be answered:

- 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 method invocation concurrency. 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 method regardless wherever the real objects are located. This property is described in section 2.3.

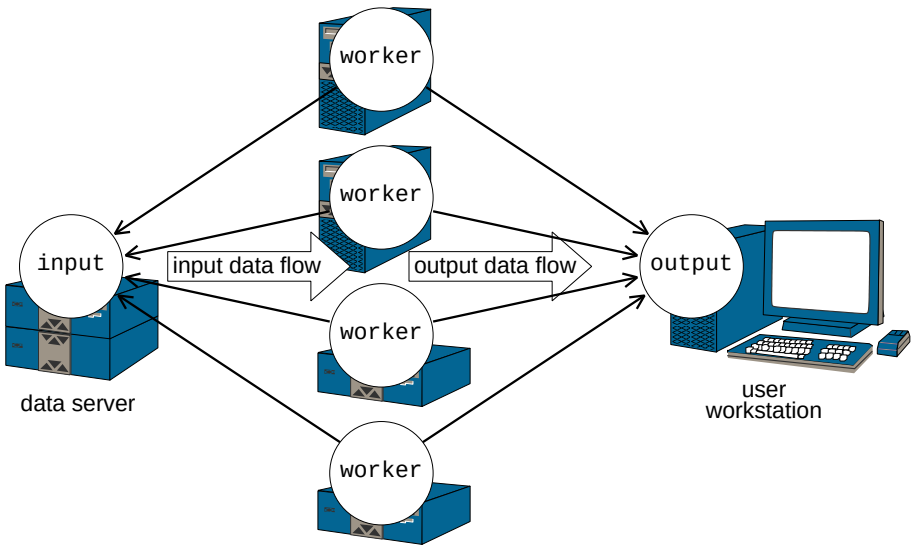
- Syntactically, invocations on parallel objects are identical to invocations on traditional sequential objects. However, parallel objects support various method invocation semantics: synchronous or asynchronous, and sequential, mutex or concurrent. These semantics are explained in section 2.4.
- Objects can be located on remote resources and in a separate address space. 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

Without further action, a parallel object lies in an inactive state. It can only be activated by receiving a method invocation request. Waiting for and accepting incoming requests at the server side are performed implicitly and transparently to the programmer. Hence, the programmer does not have to implement the object body to schedule the acceptance of method invocations. This simplifies the object execution control, allowing better integration with other software components.

## 2.3 Shareable Parallel Objects

All POP objects are shareable. Shared objects with encapsulated data provide a means for programmers to implement global data sharing in distributed environments. Shared objects can be useful in many cases. For example, figure 2.1 illustrates a scenario of using shared objects: `input` and `output` 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



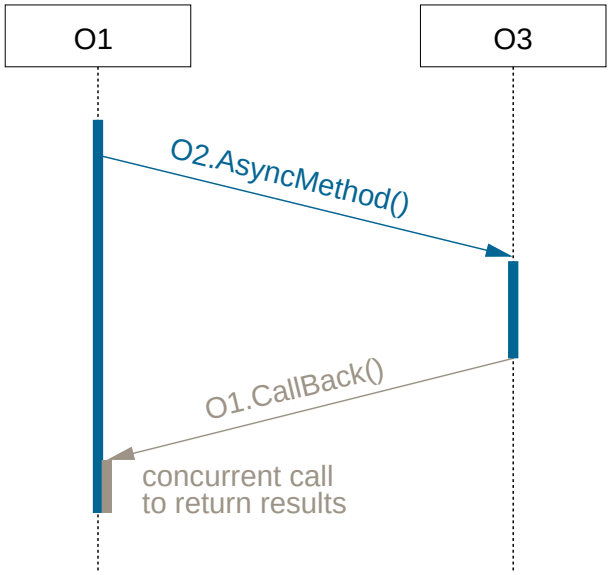

In order to share a parallel object, POP-C++ allows parallel objects to be arbitrarily passed from one place to another as arguments of method invocations. It is the runtime system, not the programmer, which is responsible for setting up the interface and managing the object references. Objects are physically destroyed only if there is no other references to them.

## 2.4 Invocation Semantics

Syntactically, method invocations on POP objects are identical to those on traditional sequential objects. However, to each method in a parallel object, one can associate different invocation semantics. All invocation semantics are specified by programmers at the design phase of parallel objects. These semantics define different behaviours at both sides of the parallel object, as follows:

- Interface semantics, the semantics that affect the caller of method invocations:
  - **Synchronous invocation**: the caller waits until the execution of the requested method on the object side is finished and returned the results. This corresponds to the traditional (remote) 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, at the time the execution returns, no computing result is available yet. This excludes asynchronous invocations from producing results. Results can be actively returned to the caller object using a callback interface, as a reference to the caller object can be passed as an argument (see figure 2.2).

**Figure 2.2** Callback method returning values from an asynchronous call




- Object-side semantics, execution semantics of methods inside each POP object:
  - **Sequential invocation**: invocations to sequential methods are executed in mutual exclusion, following the requests' arrival order. When several sequential method invocations are made on one parallel object, these requests will be served sequentially (see figure 2.3). Nevertheless, concurrent methods that have been previously started can still continue their normal execution. Sequential methods guarantee the serializable consistency of all sequential invocations inside the same object
  - **Mutex invocation**: invocations to mutex methods are executed in complete exclusion with all other methods of the same object. A request is executed only if no other invocation are running. Otherwise, the current method will be blocked until all previous invocation requests are terminated (see figure 2.3). Mutex invocations are important to synchronize concurrencies and to assure the correctness of shared data state inside the parallel object (as to implement mutual exclusive write on the same data, for example).

– **Concurrent invocation**: invocations to concurrent methods are executed concurrently if no mutex invocation is currently running or pending. All invocation instances of the same object share the same object data attributes. The concurrent invocation is important to achieve the parallelism inside each parallel object and to improve overlapping between computation and communication.

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** Example of different invocation requests

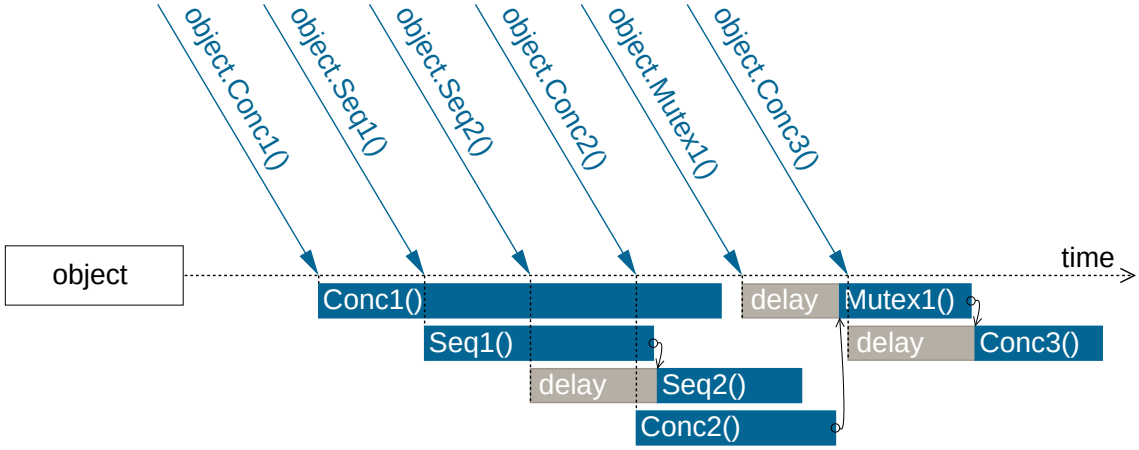



Figure 2.3 illustrates different method invocation semantics. Sequential invocation `Seq1()` is served immediately, running concurrently with `Conc1()`. Although the sequential invocation `Seq2()` arrives before the concurrent invocation `Conc2()`, it is delayed due to the current execution of `Seq1()`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

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 enable application components 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 components 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 parameterized 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

In addition to the fact that POP has a suitable programming model for the Grid, it should remain as close as possible to traditional object oriented programming. Parallel objects of the POP model generalize the sequential objects by keeping its object oriented properties such as data encapsulation, inheritance and polymorphism and by adding few new properties such as distributed shareable objects, transparent and dynamic object allocation driven by high level resource requirements and various method invocation semantics.

POP-C++ language is an extension of C++ that implements the POP model. This extension is intentionally kept as close as possible to standard C++ so that C++ programmers can easily learn to use POP-C++. Existing C++ libraries can be parallelized using POP-C++ without much effort. Changing from a sequential C++ application to 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. In order to help the POP-C++ runtime to chose a remote machine for the remote object execution, programmers can add object description information to each constructor. In order to create parallel execution, POP-C++ offers new semantics to method invocations. This new semantics are indicated through five new keywords. Synchronizations between concurrent calls are sometimes necessary, as well as event handling, and some tools are supplied for that. This chapter describes the syntax for parallel class declaration, the object description, method invocation semantics, the synchronization tool and event handling.

## 3.2 Parallel Objects

POP-C++ is an extension of C++ that implements the parallel object model as defined in chapter 2. A POP-C++ parallel object is a generalization of the sequential object. Unless the term **sequential**

**object** is explicitly specified, a parallel object is simply referred to as an object.

## 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` following 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).

All C++ classes with the following restrictions can be implemented as parallel object classes without any semantic changes:

- 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.

## Creation and Destruction

As a parallel object can be accessible concurrently from multiple distributed locations, 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.

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 3.3 will describe the POP-C++ exception mechanism.

## Parallel Class Methods

As sequential classes, parallel classes contain methods and attributes. Method accesses can be public, protected or private while attribute accesses 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 asynchronous concurrent method that returns an integer number:

```
async conc int myMethod();
```

Programmers can help the POP-C++ compiler to generate efficient code by optionally specifying which arguments to be transferred. 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 interface to the remote object implementation. Output arguments will only be transferred back to the interface after 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.
  - Pointer and array arguments are considered as both input and output.

POP-C++ automatically transfers all non-pointer user-defined types (simple types and struct types). Data pointers in C++ are ambiguous (pointer to data or starting address of an array). Therefore, programmers have to explicitly supply the number of elements pointed by that data pointer using a directive `size` in the argument information block. Programmers can also implement a specific function for marshalling and demarshalling the arguments. The `proc` directive (see section 3.2) is used to provide the name of the marshal and demarshal function. Void pointers (`void *`) cannot be used as arguments of parallel object methods.

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.

**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);
    ...
```

## Object Description

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

`@{`$expressions$`}`

An object description contains a set of resource requirement expressions. All 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 ($res_N$ 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).

**Figure 3.2** Object descriptor example

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

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 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.

## Data marshalling

When calling remote methods, the arguments must be transfered to the object being called (the same happens for returned values). In order to operate with different architectures, data is marshalled into a standard format prior to be send to remote objects, allowing different architectures to interact. All data passed is marshalled in the caller side, which means, it is converted from the native binary format to a standard format. In the callee side, data is again converted to the native binary format, or it is demarshalled.

Any scalar data, vectors, and structures of them, as well as parallel object references can be automatically marshalled. For structured data containing pointers, the programmer must supply a marshalling and demarshalling function through the directive `proc=`<*function name*> in the marshalling block in front of the argument declaration. This function is called when its associated data type is used as an argument (or return value) in a remote call.

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.

For marshalling/demarshalling complex structures, sometime, programmers need to allocate temporary memory space before serving the invocation request. This memory space will be then freed automatically by the system after the invocation finished. POP-C++ provides class `POPMemSpool` with method `Alloc` to do this temporary memory allocation.

Figure 3.3 shows an example of data marshalling in POP-C++. In this example, the programmer provides the function `marsh()` for marshalling the argument `data` of method `accelerate()` of parallel class `Engine`. The user 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).
- `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

**Figure 3.3** 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 temorary 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);
        ...
    };
```

object-side, before unpacking `val`, we need to perform temporary memory allocation using the memory spool interface provided by `tmpmem`.

## Marshalling Objects

To be able to pass objects as arguments to `parclass` method invocation, programmers must derive their classes from a POP-C++ system class of type `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 anda flag `pack` specifying if it is to serialize data into the buffer or to deserialize data from the buffer.

**Figure 3.4** 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);
    ...
};
```

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

## 3.3 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.

## 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 3.5** 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 3.5.

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.

Conditions can be waited and raised. Calls to `wait()` cause the calling thread to pause its execution until another thread triggles 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 previnously 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 3.6 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

**Figure 3.6** Using the synchronizer lock

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

the information is available. Producer threads must signal that the information is already available. Figure 3.7 is an example that shows the use of the condition.

**Figure 3.7** Using the synchronizer condition

```
class 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... */
    }
};
```

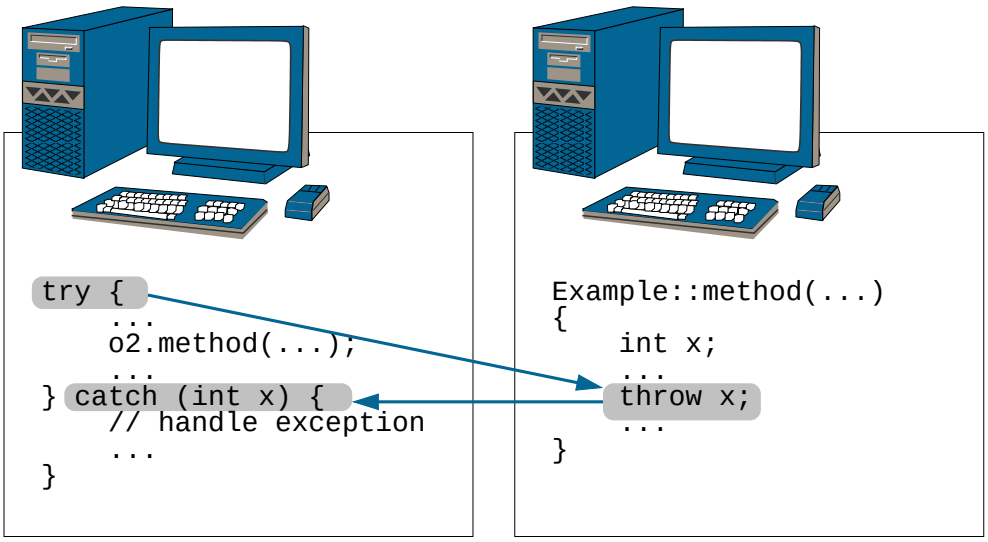
## Exceptions

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 3.8** Exception handling example




POP-C++ supports transparent exception propagation. Exceptions thrown in a parallel object will be automatically propagated back to the remote caller (figure 3.8). The current prototype POP-C++ 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 char *Extra();
   int Code();
};
```

`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.

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.

## 3.4 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 objects 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 3.9** 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 3.9 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`.

## 3.5 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 invotations.

**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;
}
```

Figure 3.10 showed 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.

**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
```

Since an MPI program requires special treatment at startup (mpirun, MPI_Initialize, etc.), users must use a POP-C++ the built-in class template `POPMPI` to create parallel object-based MPI processes. Figure 3.11 illustrates how to start, 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 decribed as follows:

```
template<class T> class POPMPI
{
 public:
  POPMPI(); //Do not create MPI process
  POPMPI(int np); // Create np MPI process of type T
  ~POPMPI();

  bool Create(int np); // Create np PI 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
};
```

## 3.6 Limitations

There is a certain number of limitations in the current implementation of POP-C++. This limitations are expected to disappear in the future, as new versions of the compiler, the library and the runtime system are released. For the current version (1.1), the limitations are:

- A parallel class cannot contain public attributes
- A parallel class cannot contain a class attribute (`static`)
- An asynchronous method cannot return a value and cannot have output parameters
- Global variables exist only in the scope of parallel objects (`@pack()` scope)
- `void` pointer parameter is not allowed
- A parallel object method cannot return a memory address
- The programmer must specify the size of pointer parameters (they are considered vectors)
- Sequential classes used as parameter must be derived from `POPBase` and the programmer must implement the `Serialize` method.
- Parameters must have the exact same type as in the method declaration, a derived object cannot be used.

CHAPTER

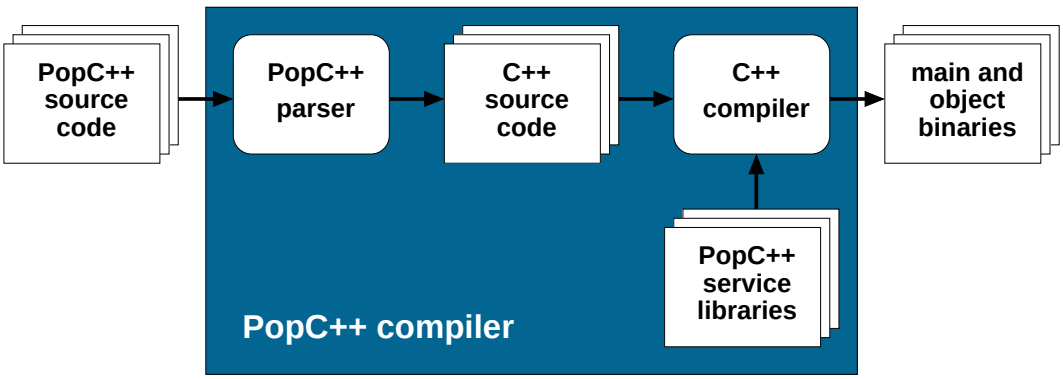
# 4 Compiling and Running



## 4.1 Compilation

POP-C++ is an extension of C++, incorporating some new keywords. Thus, standard C++ compilers cannot directly compile programs written in POP-C++. For this reason, POP-C++ incorporates a compiler which translates the POP-C++ code to the ANSI C++ code. The POP-C++ compiler generates a main executable program and several object executables. The main executable provides an entry point to start running the application. Object executables are not run directly, but they are loaded and started by the POP-C++ runtime system during the parallel objects creation. The compilation process is illustrated in figure 4.1.

**Figure 4.1** POP-C++ compilation process




The POP-C++ compiler contains a POP-C++ parser which translates the code to the ANSI C++ code. POP-C++ service libraries provide APIs for accessing various runtime services such as communication, resource discovery and object allocation, etc. An ANSI C++ compiler is used to generate binary executables from the C++ code and the service libraries.

The POP-C++ compiler is used to generate the executable modules of an application. There is always a main executable and several object executables for each application. The main executable provides an entry point to run the application. Object executables are not directly loaded, but they are used by the POP-C++ runtime system during the parallel object creation.

## 4.2 Example Program

We show in this section an example of writing a POP-C++ program in the following scenario: the user sits in front of his computer, create two remote objects. He then, from the main program, passes the interface of one object to the other. The other object will perform some operations on the object provided by the user. The object in the example is represented by an integer and the operation is to increase the integer by a value remotely stored in the other integer.

### Programming

Figure 4.2 shows the declaration of a parallel class in POP-C++. From the language aspect, this part contains the major differences between the POP-C++ and the C++. However, the example shows that POP-C++ syntax is very similar to the C++ class declaration except some new keywords (in bold letters). 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(const char machine[256]) @{ 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 but the user can also use standard C++ features such as `virtual`, `const`, or inheritance with the parallel class.

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 3.4 for the `pack` directive).

**Figure 4.3** File `integer.cc`

```
 1: #include "integer.ph"
 2:
 3: Integer::Integer(int wanted, int minp) {}
 4: Integer::Integer(const char machine[256]) {}
 5:
 6: void Integer::Set(int val) {
 7:     data = val;
 8: }
 9:
10: int Integer::Get() {
11:     return data;
12: }
13:
14: void Integer::Add(Integer &other) {
15:     data = other.Get();
16: }
17:
18: @pack(Integer);
```

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 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++.

## Compiling

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

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

## Compile the object code

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

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

**Figure 4.4** File `main.cc`

```
 1: #include "integer.ph"
 2:
 3: int main(int argc, char **argv)
 4: {
 5:     try {
 6:         Integer o1(100, 80), o2("localhost");
 7:         o1.Set(1);
 8:         o2.Set(2);
 9:         o1.Add(o2);
10:         printf("Value=%d\n", o1.Get());
11:     }
12:     catch (POPException *e) {
13:         printf("Object creation failure\n");
14:         return -1;
15:     }
16:     return 0;
17: }
```

The user has to compile 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 option `-c` (compile only) with `parocc`.

## Running

After the two executables are generated, we need to create the object map file named object.map that will be used by the code manager service. The object map file contains all mappings between object name, platform and the executable location. The executable location can be an absolute path or an URL (HTTP or FTP). We assume to compile the program on Linux machines and to put the executable on the web server at the following address:

```
http://icwww.epfl.ch/tanguyen/paroc/
```

The object map file should look like:

```
Integer Linux http://icwww.epfl.ch/tanguyen/paroc/integer.obj
```

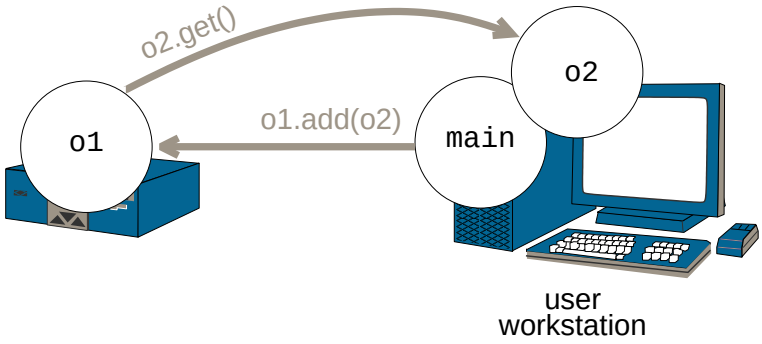
If you compile the object code for another platform (e.g. Solaris), just add a similar line to `object.map`.

Now it is ready to run the program. POP-C++ provides the command `parocrun` to do that. From the local machine, the user starts the POP-C++ `main` program by executing:

```
parocrun object.map main
```

Figure 4.5 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 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 4.5** An execution example



CHAPTER

# 5 Installation Instructions



## 5.1 Before installing

POP-C++ is built on top of several widely known software packages. It is necessary to ensure the installation of these packages prior to compile and install POP-C++. The necessary packages are:

- a C compiler (Gcc for instance);
- the Gnu Tar archiver;
- the Gnu Make utility;
- the Gnu Flex;
- the Gnu Bison;
- the Globus Toolkit (optional).

Also, before compiling and installing, some choices must be made. The system manager should make these choices and gather the following information, which will be used in the installation process:

- The compiling 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 installing directory, 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.
- 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.
- Resource topology. 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) in side 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 information about the master nodes (to which the configuring POP-C++ service will register itself to) and about the child nodes to which the configuring POP-C++ service will manage.

- The number of processors available on the resource where the POP-C++ service is representing. 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 be asked for 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.

## 5.2 Standard Installation

POP-C++ distribution uses standard Gnu tools to compile. The following commands must be given to expand the distribution tree and generate all the necessary files:

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

Please refer to the next section if you intend to install POP-C++ on top of Globus Toolkit.

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++, the following command should be used:

```
make install
```

After copying the necessary files to the chosen installation directory, a setup script is run. It asks several questions, and the information gathered before the installation should suffice to answer it. In the case it is necessary to restart the setup script, it can be done with the following command:

*installation-directory*`/sbin/paroc_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.

The current distribution of POP-C++ 1.1 supports the following features:

**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++ |

- 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

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

*installation-directory*`/sbin/paroc_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 (PAROC gateway):
sb02.eif.ch

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

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

2. Information of the local execution environment:

   - Number of processors of the local machines. 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 ParoC++ jobs that can run
concurrently(default: 1):
[Enter]

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

CONFIGURING THE RUNTIME ENVIRONMENT

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

NOTE: this information is required if you use a cluster batch job management system. An example of PBS script is provided in services/parojob.pbs of the installation tree.

```
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 PAROC 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 (SXXparoc*) 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 PAROC++ 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 PAROC 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 **paroc_setup** located in the *<installed directory>/sbin*

## 5.5 System Setup and Startup

The installation tree provides a shell setup script. It mostly 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` or `.cshrc`). The setup scripts (respectively for C-shells and Bourne shells) are:

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

Prior to any POP-C++ application execution, the runtime system must be started up. There is a script provided for that purpose, so every node must run the following command:

*installation-directory*`/sbin/SXXparoc start`

`SXXparoc` 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 `SXXparoc.globus`.

APPENDIX

# A Command Line Syntax

A.1 POP-C++ Compiler command

## A.1 POP-C++ Compiler command

```
parocc [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 ParoC++ interface codes
                          for parallel objects.
   -parclass-nobroker:    Do not generate ParoC++ broker codes
                          for parallel objects.

   -object:             Generate parallel object executable
                        (linking only)
   -parocpp:            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++:
   PAROC_LOCATION:  Directory where POP-C++ has been installed
   PAROC_CXX:       The C++ compiler used to generate object code
   PAROC_CPP:       The C++ preprocessor
   PAROC_LD:        The C++ linker used to generate binary code
   PAROC_PP:        The POP-C++  parser
```

APPENDIX

# B Runtime environment variables

Following environment vairables affect or change thhe default behaviors of the POP-C++ runtimes:

| | |
|---|---|
| PAROC_LOCATION: | Location of installed POP-C++ directory. |
| PAROC_PLUGIN_LOCATION: | Location where additional communication and data encoding plugins can be found. |
| PAROC_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 explicitely specify this information. Default value: socket://localhost:2711. |
| PAROC_HOST: | Full qualified host name of local host. This host name must be seen from outside. IP addess can also be used. |
| PAROC_PLATFORM: | The platform name of the local host. By default, the following format is used: <cpu id>-<os vendor>-<os name>. |
| PAROC_MPIRUN: | The mpirun command to start POP-C++ MPI objects. |
| PAROC_JOB_EXEC: | Script used by the job managerto submit a job to local system. |

# 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] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

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