CPPvm – C++ and PVM

Steffen Görzig

DaimlerChrysler Research and Technology 3,
Software Architecture (FT3/SA),
P.O. Box 23 60, 89013 Ulm, Germany,
steffen.goerzig@daimlerchrysler.com

Abstract. CPPvm is a C++ class library for message passing. It provides an easy–to–use C++ interface to the parallel virtual machine software PVM. CPPvm closes the gap between the design of object-oriented parallel programs in C++ and the underlying message passing possibilities of PVM. Although PVM can be used directly in C++ programs due to its C–functions, it does not support C++ specific features. CPPvm enlarges PVM with such features as classes, inheritance, overloaded operators, exception handling and streams. CPPvm also hides some details of PVM from the user and thus makes it easier to write parallel programs.

This paper describes the concepts of CPPvm. An example will explain how to transfer C++ objects between processes. CPPvm is available for many architectures, from Windows to several UNIX derivatives.

1   Introduction

Software libraries for cluster computing like the Parallel Virtual Machine (PVM, [1]) or the Message Passing Interface (MPI, [2]) mainly support procedural programming languages such as Fortran or C. Since object-oriented programming has become more and more popular in the last decade, several projects have started to develop class libraries based on existing message passing software. Examples for PVM are Para++ [3], PVM++ [4], and EasyPvm [5]. Examples for MPI are Para++, OOMPI [6], and the MPI-2 C++ bindings for MPI [7].

This paper describes CPPvm (C Plus Plus Pvm, [8]). CPPvm is a C++ message passing class library built on top of PVM. CPPvm enlarges PVM with C++ features such as classes, inheritance, overloaded operators, exception handling and streams. The main functionality of CPPvm is to pass the contents of C++ objects between several processes running in parallel. The processes can run on a network of computers with heterogenous architectures and different operating systems (e.g. Windows and the most UNIX derivatives). The parallel virtual machine connects these hosts to build just one machine. This allows a transparent message passing between all CPPvm processes running on these hosts.

2 Concepts

The fundamental concept of CPPvm is identical to that of PVM: a heterogeneous collection of hosts hooked together by a network can be used as a single large parallel virtual machine. Processes running on these hosts can become part of the virtual machine system. The processes can also spawn other processes on every host in the system and exchange data among each other.

This scenario directly leeds to a set of classes: Processes must first be linked to the parallel virtual machine. This is done by CPPvm classes for process handling which allow processes to connect to PVM and spawn child processes. "Send and receive" stream classes are furthermore needed for explicit message passing between processes. The stream classes are the transport channels for message objects. It is also possible to use message objects as distributed or mailbox objects. Figure 1 shows the hierarchy of available message classes.
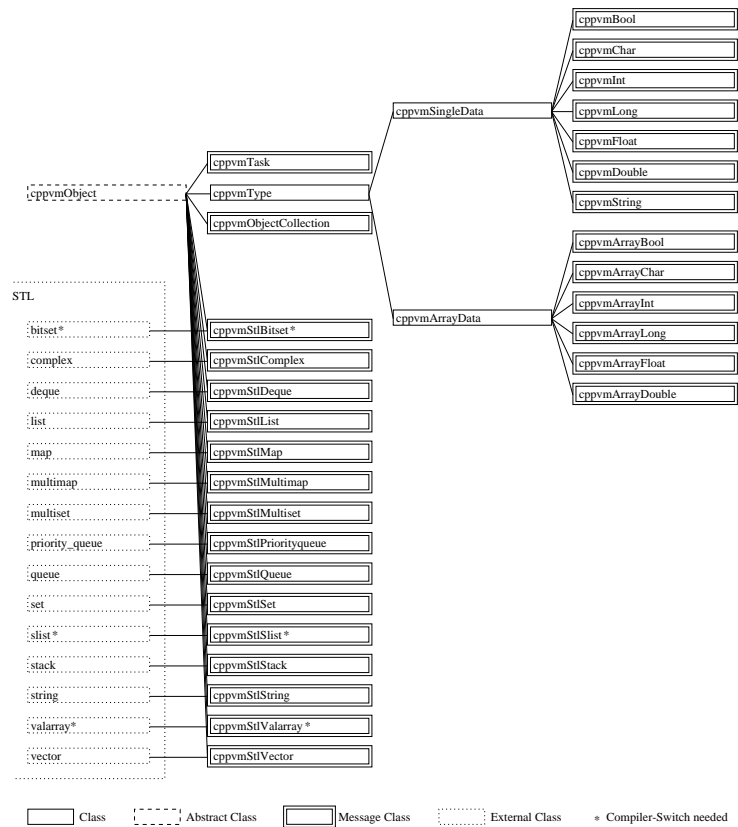
Fig. 1. Message class hierarchy. The message classes can be used for explicit message passing as well as for distributed and mailbox objects.

2.1 Explicit Message Passing

Explicit message passing is used to transfer data between objects of different processes. Processes within CPPVM usually have a parent–child relationship. For example in figure 2 process P1 has spawned process P2 on host A and process P3 on host B.
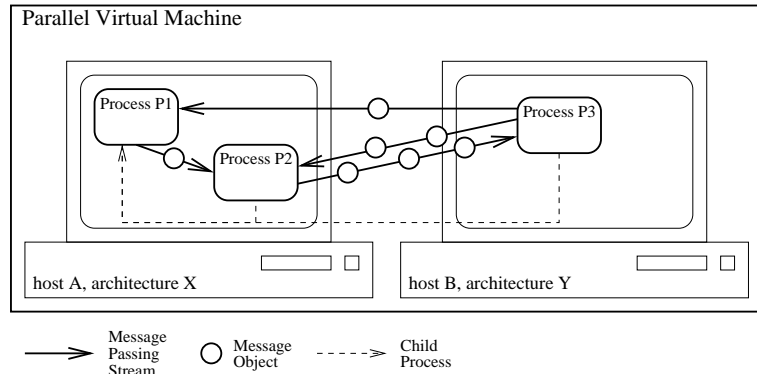

Fig. 2. Explicit Message Passing.

These relationships are used to define CPPVM message passing streams. In the example process P1 has opened a send stream to its child process P2. The message passing streams are used to send or receive C++ objects. In contrast to PVM objects can be sent with or without waiting until the other process has received the object (blocking or nonblocking). The modes of the receiving stream are blocking, nonblocking and "timeout receive". The latter tries to receive an object within a specified time range, otherwise the process continues without having received the object.

2.2 Distributed Objects

Another possibility to exchange data between processes are "distributed objects" (see figure 3). A distributed object is a C++ object which has instances in several processes.

A global data base contains one instance for every distributed object in the system. Processes can have a local instance of this object. A "read method" of the object is used to update the local object, a "write method" updates the corresponding object in the data base. When using distributed objects there is no need for send/receive streams, since the instances of a distributed object are matched by a unique identifier.
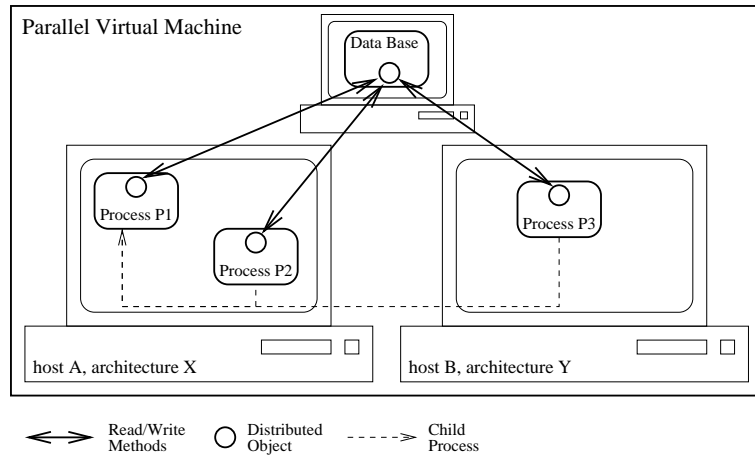
Fig. 3. Distributed Objects.

2.3 Message Mailbox Objects

Message mailbox objects are a superset of distributed objects. Mailbox objects allow to generate more than one instance of one object in the global data base (see figure 4). Whilst for distributed objects the connection between the local object and the instance in the data base is one-to-one, for message mailbox objects, the local object can randomly access all instances of an object in the data base (1:n connection).

2.4 User Defined Classes

Standard CPPvm message classes (see figure 1) should be sufficient for most kinds of parallel applications. However an interesting feature of CPPvm is the possibility to write user-defined message classes. These classes can be used for explicit message passing as well as for distributed and message mailbox objects. It is also possible to enhance existing classes with this ability, e.g. transform an existing program into a parallel program.

To create a CPPvm message class it must be derived from the class **cppvmObject**. Thereafter few modifications have to be made on this class:

- Modify/create the constructor and virtual destructor.
- Implement the virtual method `cppvmTransfer`: This method defines all message passing class variables.
- Add the macro `CPPvmMethodsDeclaration(msgtag)` to the class declaration.

The constructor of the class must call the constructor of the class **cppvmObject** (directly or indirectly). The virtual destructor is needed e.g. for deleting a collection of objects.
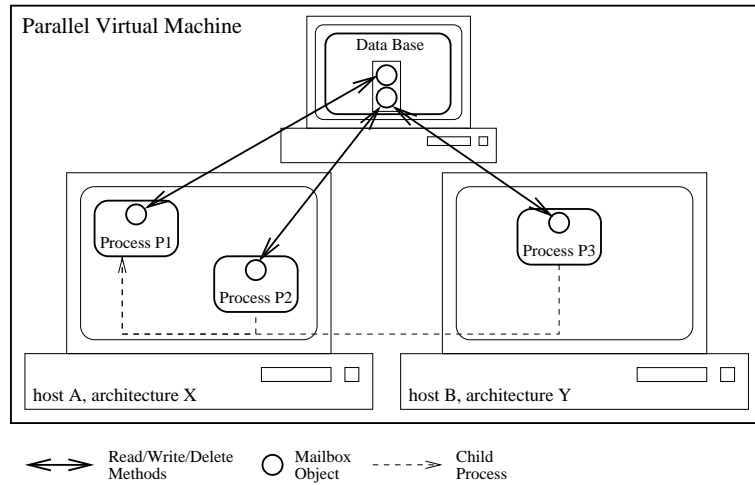
Fig. 4. Message Mailbox Objects.

The method `cppvmTransfer` specifies all data used for message passing. Only the data defined in this method is transfered. This method is called for explicit message passing as well as for distributed objects and message mailbox objects. The message passing data defined in the method `cppvmTransfer` can be:

- Standard C/C++ types (single values and arrays).
- Other CPPVM message objects derived from `cppvmObject.`
- The data of the base class.

2.5 Advanced Topics

CPPVM contains many more concepts for parallel programming than those described above. These concepts cannot be explained in detail in this paper, the following list might however help to give an impression of the possibilities opened by CPPVM:

Semaphores CPPVM includes an implementation of the semaphore concept proposed by Dijkstra [9]. Semaphores are used to synchronize parallel processes. They can also be used for critical sections. A critical section is a set of instructions which shares resources with other parallel processes. The result of a critical section can change unpredictable when these processes are running at the same time. A critical sections can be controlled by semaphores allowing only one process to enter the section at one time.

Multi-Spawn In CPPVM more than one child process can be spawned at one time. Messages can be broadcasted to all subprocesses. This feature supports architectures of parallel programs dividing a complex problem into less complex, identical subproblems.

Catchout The output of a child process can be redirected to `cout/cerr` of the master process or into a file.

Forward When using explicit message passing incoming messages can be directly forwarded to other processes.

Groups Processes can be combined to form groups. Messages can be broadcasted to all members of a group. A process can be a member of several groups simultaneously.

Context Processes can be spawned into a special context. Messages sent within one context cannot be received in another context. Therefore a context can help to avoid misleading messages. This is useful e.g. for the design of large parallel programs using old code or parallel libraries stemming from other developers.

Exceptions CPPVM uses C++ exceptions to indicate internal errors. Every error which occurs in CPPVM throws an exception object which can be caught by the user and handled individually.

Notifications Notification classes give informations about modifications of the virtual machine: adding new hosts, a host is deleted or crashed, a process exits or is killed.

Templates CPPVM also supports message passing for C++ template classes.

3 Example for Explicit Message Passing

Imagine a greeting ceremony among sportsmen called "give-me-five". The program consists of the two processes `coach` and `player` (see figure 5).
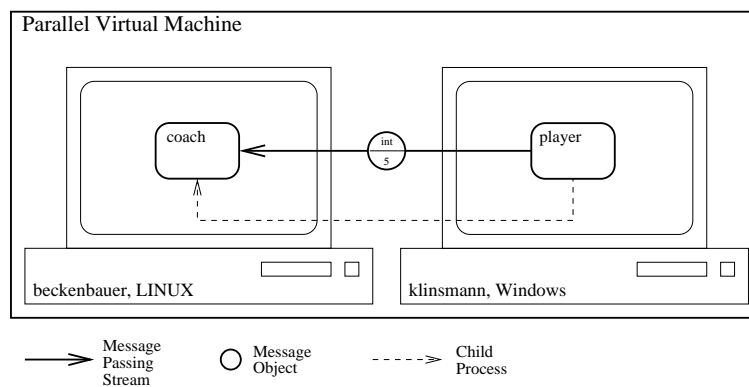
Fig. 5. Give me five!

The process `coach` is started on the host `beckenbauer` and spawns the process `player` on the host `klinsmann`. `player` sends a message to `coach`. `coach` prints the message and PVM is halted. The source code is then the following:

<div style="text-align:center">coach.cpp</div>

```
#include "cppvm.h"

int
main()
{
  int value;

  // spawn child 'player' on host
  // 'klinsmann'
  cppvmSpawnConnection child("player", "",
    PvmTaskHost, "klinsmann");

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
    CPPvmRBchild);

  cout << "coach: give me five!" << endl;

  // receive value from player
  recStrm >> value;

  cout << "player: " << value << endl;

  // halt the virtual machine
  child.halt();

  return 0;
}
```

<div style="text-align:center">player.cpp</div>

```
#include "cppvm.h"

int
main()
{
  int value=5;

  // connect to pvm
  cppvmConnection pvmConn;

  // send descriptor
  // (nonblocking send
  //  to parent process)
  cppvmSendStream sendStrm(
    pvmConn, CPPvmSNBparent);

  // send value
  sendStrm << value;

  return 0;
}
```

For explicit message passing following types can be used:

- all CPPvm message classes (see figure 1)
- the standard C++ types bool, char, double, float, int and long as well as constants[1]
- the standard template library (STL) classes bitset, complex, deque, list, map, multimap, multiset, priority_queue, queue, set, slist, stack, string, valarray, and vector

4 Conclusion

Object-orientation is state-of-practice in the field of programming. CPPvm was designed to support object-oriented programming in C++ for cluster computing.

CPPvm is based on the Parallel Virtual Machine (PVM) and is published under the GNU Library General Public License (LGPL) [8]. As shown, CPPvm enlarges PVM with C++ features as classes, inheritance, overloaded operators, exception handling and streams. CPPvm also hides some details of PVM (e.g. starting PVM daemons or adding hosts) from the user and thus makes it easier to write parallel programs. CPPvm is available for many architectures, from Windows to several UNIX derivatives. CPPvm allows to:

---

[1] Constants can of course only be sent and not received because they cannot change their values.

- combine a heterogenous collection of computers
- spawn and kill processes dynamically
- detect failed processes and hosts
- send/receive C++ objects
- use distributed C++ objects
- use C++ objects together with a message mailbox
- write user-defined message C++ classes
- use C++ templates
- use standard template library (STL) classes
- use semaphores
- use CPPvm together with existing PVM software

CPPvm has a very detailed documentation in several formats (Postscript, PDF and HTML). Many examples – from the very simple "hello world" program to more complex application/server programs – help the user to make his way from his very first parallel steps to the innermost secrets of MIMD programming.

References

1. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam: *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994, http://www.epm.ornl.gov/pvm/pvm_home.html/
2. *MPI: Message Passing Interface*, http://www.erc.msstate.edu/labs/hpcl/projects/mpi/
3. O. Coulaud and E. Dillon: "PARA++ : C++ Bindings for Message Passing Libraries", in *The EuroPvm '95 Users Meeting*, Lyon, France, http://www.loria.fr/projets/para++/
4. *PVM++*, http://goethe.ira.uka.de/~wilhelmi/pvm++/
5. *EasyPvm*, http://www.brunel.ac.uk/~mepghfb/pvm_c++_wrapper.htm
6. J. M. Squyres, B. C. McCandless, and A. Lumsdaine: "Object Oriented MPI: A Class Library for the Message Passing Interface", in *The 1996 Parallel Object-Oriented Methods and Application Conference (POOMA '96)*, Santa Fe, New Mexico
7. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. Message Passing Interface Forum, http://www.mpi.nd.edu/research/mpi2c++
8. S. Görzig: *CPPvm: C++ Interface to PVM (Parallel Virtual Machine)*, 1999. http://www.informatik.uni-stuttgart.de/ipvr/bv/cppvm
9. E. W. Dijkstra: "The Structure of the THE Multiprogramming System", in *Commun. of the ACM 11*, pp. 341–346, May 1968