CPPvm – Parallel Programming in C++

Steffen Görzig
DaimlerChrysler Research and Technology 3,
Software Architecture (FT3/SA),
P.O. Box 23 60, 89013 Ulm, Germany.
E-mail: 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, 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. *Examples explain how to apply two main message passing methods: the transfer of C++ objects between processes and the use of distributed C++ objects. An extension of* CPPVM *enables user defined classes. This simplifies, for example, the transformation of existing C++ programs into parallel programs running on a computer cluster.*

1 Introduction

In the past most software libraries for cluster computing like the Parallel Virtual Machine (PVM, [7]) or the Message Passing Interface (MPI, [2]) focused on procedural programming languages such as Fortran or C. As the popularity of object-oriented programming has increased in the last decade, several projects have started to develop class libraries based on existing message passing software. Examples for PVM are Para++ [5], PVM++ [3], and EasyPvm [1]. Examples for MPI are Para++, OOMPI [9], and the MPI-2 C++ bindings for MPI [4].

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, and streams.

The fundamental concept of CPPVM is identical to that of PVM: a heterogeneous collection of hosts with heterogenous architectures and different operating systems (e.g. Windows and the most UNIX derivatives) 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. Additionally processes can spawn other processes on every host in the system and then exchange data among each other.

The main functionality of CPPVM is to enable the exchange of C++ objects between several processes running in parallel (explicit message passing). Another possibility to share data among processes are distributed objects which are instantiated in several processes. A large set of classes can be used for these kinds of message passing. 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 objects. Therefore, by enhancing existing classes, a program is easily transformed into a parallel program running on a computer cluster.

2 Explicit Message Passing

Explicit message passing is used to transfer data between objects of different processes. Usually, the processes within CPPVM have a parent–child relationship. Streams are used to pass messages between the processes. Data can be send blocking or nonblocking, i.e. the sending process waits until the object has been transmitted to the receiver or proceeds without waiting. The modes of the receive stream are blocking, nonblocking and ”timeout receive”. A timeout receive tries to receive an object for a specified time, otherwise the process continues without having received the object.

Imagine two soccer players passing a ball. The parallel program consists in two processes, `forward` and `midfield` (see figure 1).

The process `forward` is started on the host `klinsmann` and spawns the process `midfield` on the host

`matthaeus`. `midfield` sends a message to `forward`. `forward` prints the message and the PVM is halted. The source code is then the following:

forward.cpp

```
#include "cppvm.h"
#include <string>
using namespace std;

int
main()
{
  string item;

  // spawn child 'midfield'
  // on host 'matthaeus'
  cppvmSpawnConnection child("midfield",
    "", PvmTaskHost, "matthaeus");

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

  // receive item from midfield
  recStrm >> item;

  cout << "received: " << item << endl;

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

  return 0;
}
```

midfield.cpp

```
#include "cppvm.h"
#include <string>
using namespace std;

int
main()
{
  string item="ball";

  // connect to pvm
  cppvmConnection pvmConn;

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

  // send item
  sendStrm << item;

  return 0;
}
```

In contrast to PVM there is no need to start the local daemon or to add the host `matthaeus` to the virtual machine: this is encapsulated by CPPVM. Additionally CPPVM supports more data types for message passing than PVM does. In the example a C++ object of the standard template library (STL) class `string` is used. Other supported types for explicit message passing are:

- predefined CPPVM message classes derived from the base class cppvmObject (see figure 2)
- 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

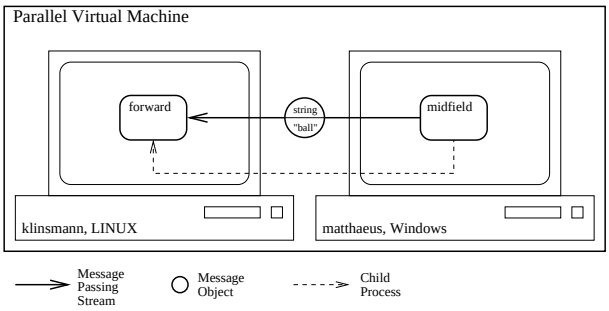

Figure 1. Pass the ball!

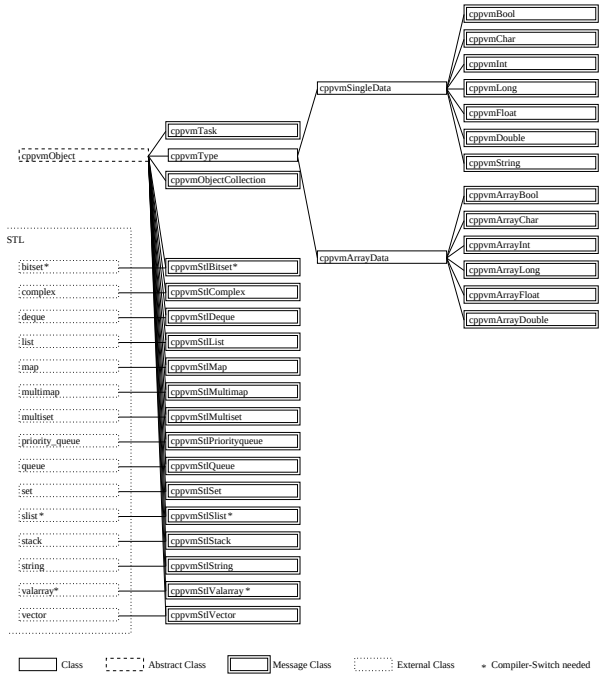

Figure 2. CPPVM **message classes.**

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

3 Distributed Objects

Distributed CPPVM objects are objects which can be accessed from every process connected to the PVM. A distributed object consists in a local object which can be synchronized with an object in the PVM message mailbox.

Here is an example of how to transfer the score of a match from the control-room to the score-board:

panel.cpp

```
#include "cppvm.h"

int
main()
{
  // spawn child process 'display'
  cppvmSpawnConnection child("display", "",
    PvmTaskHost, "score-board");

  cppvmInt teamRed;
  cppvmInt teamBlue;

  // connect to message mailbox
  teamRed.cppvmDistObj("red");
  teamBlue.cppvmDistObj("blue");

  int i;
  for(i=0; i<500; i++){
    teamRed = 0;
    teamBlue = i;
    // update mailbox object
    teamRed.cppvmWriteMboxInfo();
    teamBlue.cppvmWriteMboxInfo();
  }

  // halt the virtual machine
  child.halt();
  return 0;
}
```

display.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // score objects
  cppvmInt teamRed;
  cppvmInt teamBlue;

  // connect to message mailbox
  teamRed.cppvmDistObj("red");
  teamBlue.cppvmDistObj("blue");

  int i;
  for(i=0;i<500;i++){
    // update the local data
    teamRed.cppvmReadMboxInfo();
    teamBlue.cppvmReadMboxInfo();
  }

  return 0;
}
```

The process `panel` spawns the child process `display`. The process `panel` registers the objects `teamRed` and `teamBlue` as distributed integer objects using the key names `red` and `blue`. The registration is done with the method `cppvmDistObj`. When calling this method an object with the given name is added to the message mailbox (if it does not already exist). `display` connects to the PVM and registers two integer objects using the same key names. Thereafter the local integer objects are connected to each other via the message mailbox.

In the example, `panel` updates the mailbox objects (see figure 3). This is done by calling the method `cppvmWriteMboxInfo`. The process `display` reads the objects from the mailbox with the method `cppvmReadMboxInfo`. These methods have no effect if the object has not been registered as a distributed object. The registration of an object as distributed object does not exclude explicit message passing. The object can still be sent/received as described in section 2.

4 User Defined Classes

In the score-board example above two variables (`teamRed` and `teamBlue`) are used to transfer data among processes. In order to show how CPPVM is enhanced by user-defined classes, a score class is defined which encapsulates the score of the red and blue team. The implementation is given below:

score.h

```
#ifndef _SCORE_INCLUDE
#define _SCORE_INCLUDE

// base class 'cppvmObject'
#include "pvm_obj.h"

class score : public cppvmObject{
public:

  score() : cppvmObject() {
    teamRed=0;
    teamBlue=0;
  }
  virtual ~score(){}

  int teamRed;
  int teamBlue;

  // message tag declaration
  CPPvmMethodsDeclaration(123)

  // transfer method
  virtual void cppvmTransfer(){
    cppvmTransferInt(&teamRed);
    cppvmTransferInt(&teamBlue);
  }
};
#endif
```

The class score is derived from the class cppvmObject. Therefore, `pvm_obj.h` has to be included. The constructor of score calls the constructor of its base class cppvmObject in order to initialize the CPPVM facilities. The message tag of the class is set to 123 (macro `CPPvmMethodsDeclaration`). This tag is used to determine the type of incoming messages and should be unique to avoid misleading messages. The method `cppvmTransfer` defines the class variables to be transfered during message passing. Within

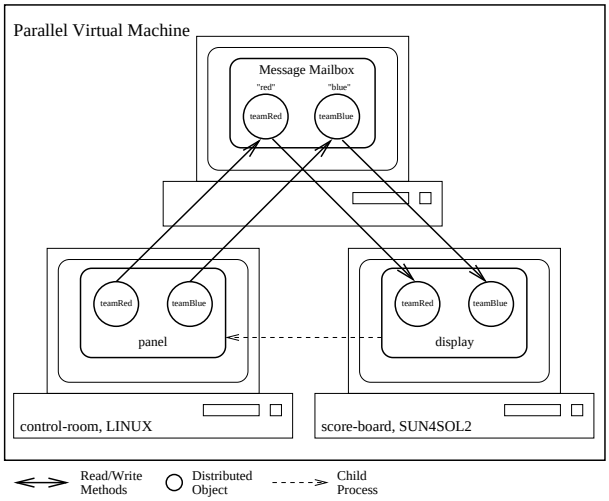

Figure 3. The score of the match.

this method the function `cppvmTransferInt` is used to transfer the variables `teamRed` and `teamBlue`.

The score class can then be used for all kinds of CPPVM data transfer: for explicit message passing as well as for distributed objects (see examples in 2 and 3).

5 Conclusion

CPPVM was designed to support object-oriented programming in C++ for cluster computing. As shown, CPPVM enlarges PVM with C++ features as classes, inheritance, overloaded operators, 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.

Beside the described explicit message passing, distributed objects, and user defined classes, CPPVM contains many more concepts for parallel programming. Message mailbox objects, for example, are a superset of distributed objects. Mailbox objects allow to generate more than one instance of an object in the global database. When using explicit message passing incoming messages can be directly forwarded to other processes. CPPVM includes an implementation of the semaphore concept proposed by Dijkstra [6]. Broadcast to process groups is available as well as multicast to selected processes. The output of a child process can be redirected to `cout/cerr` of the master process or into a file. 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. CPPVM uses C++ exceptions to indicate internal errors, which can be caught by the user and handled individually. Notification classes provide information about modifications of the virtual machine. CPPVM also supports message passing for C++ template classes. A complete description of all features is given in a detailed documentation in several formats (Postscript, PDF, and HTML; [8]).

CPPVM is based on the Parallel Virtual Machine (PVM) and is published under the GNU Library General Public License (LGPL) [8]. CPPVM is available for many architectures, from Windows to several UNIX derivatives.

References

[1] *EasyPvm*. http://www.brunel.ac.uk/~mepghfb/pvm_c++_wrapper.htm.

[2] *MPI: Message Passing Interface*. http://www.erc.msstate.edu/labs/hpcl/projects/mpi/.

[3] *PVM++*. http://goethe.ira.uka.de/~wilhelmi/pvm++/.

[4] *MPI-2: Extensions to the Message-Passing Interface*, July 1997. Message Passing Interface Forum, http://www.mpi.nd.edu/research/mpi2c++.

[5] 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++/.

[6] E. W. Dijkstra. The Structure of the THE Multiprogramming System. In *Commun. of the ACM 11*, pages 341–346, May 1968.

[7] 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/.

[8] S. Görzig. *CPPvm: C++ Interface to PVM (Parallel Virtual Machine)*, 1999. http://www.informatik.uni-stuttgart.de/ipvr/bv/cppvm.

[9] 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*.