ClassdescMP: Easy MPI Programming in C++

Russell K. Standish[1,2] and Duraid Madina[2]

[1] School of Mathematics
[2] High Performance Computing Support Unit
University of New South Wales,Sydney, 2052,Australia
{R.Standish,duraid}@unsw.edu.au
http://parallel.hpc.unsw.edu.au/classdesc

Abstract. ClassdescMP is a distributed memory parallel programming system for use with C++ and MPI. It uses the Classdesc reflection system to ease the task of building complicated messages to be sent between processes. It doesn't hide the underlying MPI API, so it is an augmentation of MPI capabilities. Users can still call standard MPI function calls if needed for performance reasons.

1 Classdesc Reflection

MPI is an *application programming interface* (API - in other words library of functions calls) for passing data from one unix process to another. The processes may be running on the same computer, or completely distinct computers, so this system provides a means of implementing *distributed memory parallel processing*.

MPI has been used to great success in a variety of Engineering and Scientific codes where large arrays of the same type of data (eg floating point numbers) need to be exchanged between processes. However, with object oriented codes, one really needs to send *objects* (which may well be compound) between processes. MPI does provide the `MPI_Type` functionality, which allows the description of compound structures to be built up, however this is difficult to use, and doesn't handle the case where the structure contains references to other objects (eg pointers).

Object *reflection* allows straightforward implementation of serialisation (i.e. the creation of binary data representing objects that can be stored and later reconstructed), binding of scripting languages or GUI objects to 'worker' objects and remote method invocation. Serialisation, for example, requires knowledge of the detailed structure of the object. The member objects may be able to serialised (eg a dynamic array structure), but be implemented in terms of a pointer to a heap object. Also, one may be interested in serialising the object in a machine independent way, which requires knowledge of whether a particular bitfield is an integer or floating point variable.

Languages such as Objective C give objects reflection by creating class objects and implicitly including an *isa* pointer in objects of that class pointing to the class object. Java does much the same thing, providing all objects with the

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 896–902, 2003.

native (i.e. non-Java) method `getClass()` which returns the object's class at runtime, as maintained by the virtual machine.

When using C++, on the other hand, at compile time most of the information about what exactly objects are is discarded. Standard C++ does provide a run-time type enquiry mechanism, however this is only required to return a unique signature for each type used in the program. Not only is this signature be compiler dependent, it could be implemented by the compiler enumerating all types used in a particular compilation, and so the signature would differ from program to program!

The solution to this problem lies (as it must) outside the C++ language per se, in the form of a separate program (classdesc)[2,5] which parses an input program and emits function declarations that know about the structure of the objects inside. These are generically termed *class descriptors*. The class descriptor generator only needs to handle class, struct and union definitions. Anonymous structs used in typedefs are parsed as well. What is emitted in the object descriptor is a sequence of function calls for each base class and member, just as the compiler generated constructors and destructors are formed. Function overloading ensures that the correct sequence of actions is generated at compile time.

Once a class definition has been parsed by classdesc, and the emitted class descriptors included into the original program, an object of that class can be serialised into a buffer object using the `<<` operator. The `>>` operator can be used to extract the value back.

```
#include "foo.h"  //foo class definition
#include "foo.cd" //foo class descriptor
...
pack_t buf;
foo a=...;
bar b=...;
foo c;
buf << a << b;  // serialise a into buffer.
buf >> c;  // unpack buffer into c. Now c == a.
```

Once a buffer is packed, it can be saved to a file, or transferred over a network connection by using its public members `data`, which points to the data, and `size` which gives the number of bytes currently in the buffer.

A variant of `pack_t` is the `xdr_pack` type, which uses the XDR library to represent the objects in a machine independent form. This is needed if the receiver of the serialised object is of a different architecture (eg endianess, or wordsize) to the sender.

The only real reason for using `MPI_Type` in MPI is to support messages of compound objects between differing processors in a heterogenous cluster. `xdr_pack` gives heterogenous cluster support "for free".

2 MPIbuf

`MPIbuf` is a derived type which inherits from `pack_t` (or `xdr_pack` if the `HETERO` flag is set). Similar to the notion of *manipulators* with the standard iostream library, `MPIbuf` has manipulators, such as `send()`:

```
buf << a << b << send(1);
```

which sends `a` and `b` to process 1.

Process 1 can receive this message with:

```
buf.get() >> a >> b;
```

Other manipulators and methods of MPIbuf implement collective operations such as broadcast, gather and scatter:

```
buf << a << b << bcast(0) >> a >> b;
```

suffices to broadcast the values of `a` and `b` to all processors.

By default, an `MPIbuf` uses the `MPI_COMM_WORLD` communicator, which includes all processes started by the MPI system. Collective operations can be restricted to subsets of processes by defining an MPI *communicator* object using `MPI_Comm_create` and assigned to the `Communicator` member of the `MPIbuf` object.

3 MPISPMD

An MPI program requires a certain amount of structure to just to setup the processes, and tear them down at the finish, before any real coding starts.

A typical MPI program might do the following:

```
main(int argc,char**argv)
{
  MPI_Init(&argc,&argv);
  MPI_Comm_size(&nprocs);
  MPI_Comm_rank(&myid);
  ... computation ...
  MPI_Finalize();
}
```

One problem that frequently occurs is forgetting to call `MPI_Finalize()`, which needs to be called from all processes. This may occur because of an error condition, or otherwise abnormal exit from the program. This is a problem, as typical MPI implementations have resources such as shared memory segments that need freeing, and the slave processes need to be terminated. On a shared high performance computing system, stray processes consume CPU resources that may be better employed by other users.

`MPISPMD` arranges for `MPI_Finalize()` to be called from its destructor. This has two advantages: `MPI_Finalize()` will be called as soon as the `MPISPMD` leaves its scope. Also, the destructor will be called if an exception is raised, leading to cleaner error handling. Unfortunately, if the program terminates from a conventional low level call to `abort()`, the cleanup will not take place.

The above piece of code can be replaced by:

```
main(int argc, char** argv)
{
  MPISPMD C(argc,argv);
  ... computation ...
}
```

The number of processes and processor ID can be queried from MPISPMD::nprocs and MPISPMD::myid respectively. Note that you should not declare the MPISPMD object as a global variable, as some MPI implementations object to `MPI_Finalize()` being called after `main()` exits. If you need to refer to the MPISPMD object from global scope, declare a global pointer to MPISPMD, and initialise this from with `main()`:

```
MPISPMD *C;
main(int argc, char** argv)
{
  MPISPMD comp(argc,argv); C=&comp;
  ...
```

4 MPIslave

MPISPMD gives rather minimal support, as the SPMD programming model is already implicit in MPI. Another programming model which MPI is often put to is master-slave processing. Setting up the structure of a master-slave program is very tedious and error prone. The `MPIslave` class is designed to make master-slave algorithms simple to program.

When a `MPIslave` object is instantiated, a slave "interpreter" object is instantiated on each process to receive messages from the master. As `MPIslave` needs to know the type of object to be instantiated on the slave processes, it is implemented as a template, with the type of slave object passed as the template parameter.

A message sent to the slave process starts with a method pointer of type: `void (S::*)(MPIbuf&)` where `S` is the slave object type, followed by the arguments to be passed. That method of the slave object is then called, with the arguments passed through `MPIbuf` argument, and any return values also passed through `MPIbuf` argument:

```
struct S
{
  void foo(MPIbuf& args)
```

```
  {
    int x,y,r;
    args >> x >> y;
    ...
    args.reset() << r;
  }
};

main(int argc, char** argv)
{
  MPIslave<S> C;
  MPIbuf buf;
  int x=1, y=2;
  buf << &S::foo << x << y << send(1);
}
```

When the `MPIslave` object is destroyed on the master process, it arranges for all the slave objects to be `MPI_Finalized()` and destroyed also.

`MPIslave` also has features for managing a pool of idle slaves:

```
MPIslave<S> C(argc,argv);
vector<job> joblist;
for (int p=1; p<C.nprocs && p<joblist.size(); p++)
   C.exec(C << &S::do_job << joblist[p]);
while (p<joblist.size())
  {
    process_return(C.get_returnv());
    C.exec(C << &S::do_job << joblist[p++]);
  }
while (!C.all_idle())
  process_return(C.get_returnv());
```

5 Performance

Serialisation within classdesc is very efficient, as the relevant class descriptor functions are all inlined. The performance is comparable with that of simply copying the data into the buffer object. Using the MPIbuf mechanism incurs the overhead of copying into, and back out of an extra buffer, which is particularly wasteful if only one object is being sent. Clearly, within performance critical code, Classdesc MPIBuf operations may be replaced manually by an equivalent sequence of `MPI_Send()`s and `MPI_Recv()`s. As with any optimisation technique, this should be performed after profiling the code to ensure benefit.

At the time of writing, Classdesc has only been deployed in one production code (Eco-Tierra) [4,3]. Figure 1 shows the speedup curves on two machines: *Napier*, a 28 processor SGI Power Challenge, which is now a rather dated SMP

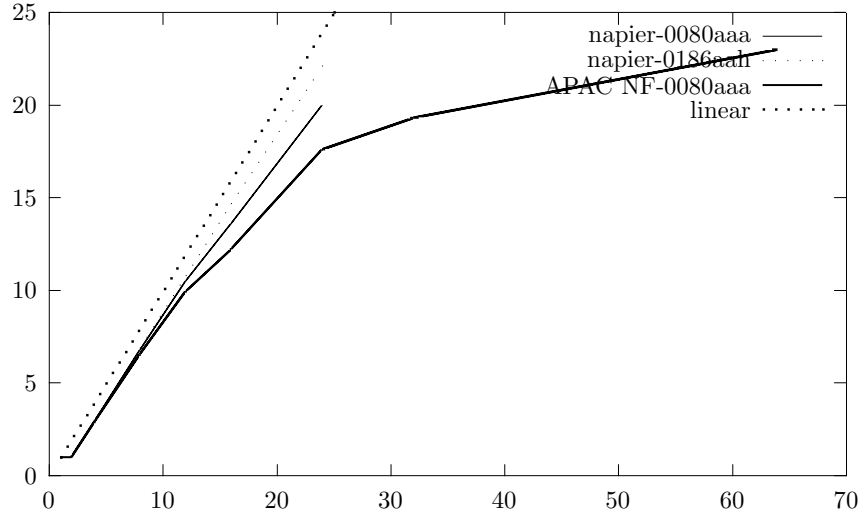

Fig. 1. Speedup curves for Eco-Tierra single site entropy jobs. Two organisms (0080aaa and 0186aah) were run on Napier, and one (0080aaa) on the APAC National Facility.

architecture, and the *APAC National Facility*, a Compaq SC cluster, consisting of 4 processor nodes coupled by a Quadrics switch.

0186aah is a larger job than 0080aaa, so has more parallelism. It is still scaling linearly at 24 processors on Napier. The constant offset from linear scaling is due to the fact that the master thread does very little, if any computation. For some reason, the 0186aah job is not running correctly on the APAC NF, so no benchmark results are available.

With 0080aaa, good scaling is obtained up to 24 processors, but on the APAC NF performance falls off after that. This is most likely an effect of the master process being unable to keep slaves busy. However, the scaling discrepancy between Napier and the APAC NF at lower process numbers indicates that network performance is partially limiting. It should be borne in mind though that at 24 processes, the APAC NF is around 6 times faster than Napier!

6 Discussion

Classdesc was originally developed as part of the *EcoLab* package to allow it to handle arbitrary agent-based models[5]. It has been used in a project studying combinatorial spacetimes[1], and in several simulations using Graphcode, a C++ framework for agent-based simulation. Graphcode considers agents to be the vertices of arbitrary hypergraphs and is equipped with a graph partitioner to allow for the efficient execution of agent-based simulations on parallel computers. To

achieve this, Graphcode uses Classdesc internally, so that objects may be serialized for transmission between cooperating processes. ClassdescMP is a natural outgrowth of Classdesc, and has been successfully employed in the Eco-Tierra project[4,3].

ClassdescMP is distributed as part of the Classdesc package, available from http://parallel.hpc.unsw.edu.au/classdesc. It should compile and be usable with any ANSI standard C++ compiler. It has been tested on a variety of operating systems, including most Unices, Windows under Cygwin and Mac OS X.

Acknowledgements. The development of Classdesc and ClassdescMP was made possible with a grant from the Australian Partnership of Advanced Computing. Benchmark results were obtained on both APAC and ac3 facilities.

References

1. Duraid Madina. Topology and complexity: From automata to agents. In Namatame et al., editors, *Complex Systems '02*, pages 141–147. Chuo University, September 2002. also to appear in *Complexity International*.
2. Duraid Madina and Russell K. Standish. A system for reflection in C++. In *Proceedings of AUUG2001: Always on and Everywhere*. Australian Unix Users Group, 2001.
3. R. K. Standish. Open-ended artificial evolution. *International Journal of Computational Intelligence and Applications*, 2003. (accepted).
4. Russell K. Standish. Some techniques for the measurement of complexity in Tierra. In Dario Floreano, Jean-Daniel Nicoud, and Francesco Mondada, editors, *Advances in Artificial Life: 5th European Conference, ECAL 99*, volume 1674 of *Lecture Notes in Computer Science*, page 104, Berlin, 1999. Springer.
5. Russell K. Standish. Ecolab4. *Complexity International*, 8, 2001.