

The Meat and Bones of Message Passing

Michael Gauckler, michael.gauckler@quantcatalyst.com
Daniel Egloff, daniel.egloff@quantcatalyst.com

September 20, 2006

Abstract

The following paper describes the *skeleton* and *content* idea behind Boost MPI. The idea is to make it easy to optimize the interprocess communication for parallel programs that transmit the same data structures many times. Boost MPI grew out of a project at Zurich Cantonal Bank where we implemented a large distributed Monte Carlo simulation to calculate the loss distribution of credit portfolios. It is now released as part of the Boost C++ libraries Gregor and Troyer (2005).

Motivation

The scalability and the performance of parallel programs depend heavily on the efficiency of their inter-process communication code. Certain algorithms, such as Monte-Carlo simulations or PDE solvers operating on a mesh, need to transmit the same data structures, filled with different data, many times. Optimizing the repeated transmission of the same data structures is therefore a key requirement for an efficient implementation of such algorithms.

One optimization is to send the data directly through the network into the desired memory location of the receiving process without using an intermediate buffer to align the data to a contiguous sequence. This avoids the cost of copying the data and the memory for keeping it at a second location. Memory demanding applications such as for instance block sampling algorithms profit from economizing on the intermediate buffer because in the best case, they can allocate data structures of twice the size.

This also eliminates the need to reallocate the data structure for every transmission, which is highly inefficient. It also reduces memory fragmentation.

To address these optimization the MPI standard introduces `MPLDatatypes`, see MPI-Forum (1996). Unfortunately, MPI datatypes—particularly the user-defined MPI datatypes—are hard to use and error-prone.

The Boost MPI lets one take advantage of these optimization without the burden of dealing with the complexity of user-defined MPI data types by providing a higher level of abstraction while handling the datatypes transparent to the user. The use of the Boost MPI will therefore make it easier and less error-prone to apply the optimizations proposed by the MPI standard.

Standard Solution

MPI derived datatypes let one define arbitrary collections of non-contiguous and non-homogeneous data in memory, which allows sending and receiving non-contiguous data in a single MPI function call. A thorough treatment of the subject can be found in Snir, Otto, Huss-Lederman, Walker and Dongarra (1996, Chapter 3, “User-Defined Datatypes”).

The object’s memory layout is represented by a so-called *Typemap*. A *Typemap* defines a list of types and displacements from a base address.

$$\text{Typemap} = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

The associated type signature contains only the sequence of types.

$$\text{Typesig} = \{type_0, \dots, type_{n-1}\}.$$

The *Typemap* describes the type and location of data in memory and lets one send it to a remote process, given that the *Typemap* on the sending process is compatible with the *Typemap* on the receiving side. Two *Typemaps* are compatible if their associated *Typesigs* are the same.

The user-defined data types can be used to define complex data structures. The types of the data items are described by primitive types (plain old data types like integer, float, character) or user-defined data types.

The offsets of the data items are defined by the derived data type routines which return a user-defined data type of the given layout. Examples of such routines and their C and C++ bindings are:

- `MPL_Type_contiguous(int count, MPL_Datatype oldtype, MPL_Datatype *newtype)`
`MPI::Datatype MPI::Datatype::Create_contiguous(int count) const`
Returns a datatype that represents the concatenation of count instances of oldtype.
- `MPL_Type_vector(int count, int blocklength, int stride, MPL_Datatype oldtype, MPL_Datatype *newtype)`
`MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength, int stride) const`
Returns a datatype that represents equally spaced blocks.
- `MPL_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPL_Datatype *array_of_types, MPL_datatype *newtype)`
`MPI::Datatype MPI::Datatype::Create_struct(int count, const int array_of_blocklengths[], const MPI::Aint array_of_displacements[], const MPI::Datatype array_of_types[])`
Returns a datatype that represents count blocks. Each is defined by an entry in `array_of_blocklengths`, `array_of_displacements` and `array_of_types`.

As one can see from the signatures the C++ versions just provide a slightly different syntax but do not add more comfort or functionality over the C bindings. To transfer a data structure from one process to another the steps described in the following table are necessary:

| Step | Sending process | Receiving process |
|------|---|--|
| | create data structure and fill it with payload | create empty data structure |
| I | send size information | receive size information and resize data structure |
| II | describe type and offset of data structure by a <i>Typemap</i> and send payload | describe type and offset of data structure by a <i>Typemap</i> and receive payload |

The following example illustrates the sending and receiving of a data structure consisting of an array of four instances of the class `gps_position`. A `gps_position` contains two integers for degrees and minutes and a float for seconds, it is depicted in Figure 1.

```
class gps_position
{
private:
```

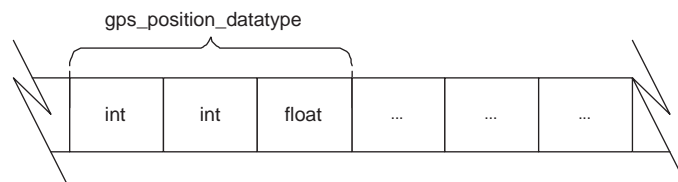


Figure 1: GPS datatype

```

    int degrees;
    int minutes;
    float seconds;
public:
    gps_position(){};
    gps_position(int d, int m, float s) :
        degrees(d), minutes(m), seconds(s)
    {}
}

```

To send an array of four `gps_position` we would first create a `MPI_Datatype` for a single GPS position and use it as a building block for `MPI_Type_contiguous`.

```

/* Initialize MPI and retrieve the rank of the process within MPI_COMM_WORLD */

gps_position      pos[4]; /* The size of 4 is hard coded and therefore
                           known to the sending and receiving process.
                           In a more general setup this information
                           needs to be transmitted separately. */

MPI_Datatype      gps_position_datatype;
MPI_Datatype      gps_position_array_datatype;
MPI_Datatype      type[2] = {MPI_INT, MPI_FLOAT};
int               blocklen[2] = {2,1};
MPI_Aint          disp[2] = {0,2*sizeof(double)};

MPI_Type_struct(2 /* count */, blocklen, disp, type, &gps_position_datatype);
MPI_Type_contiguous(4 /* count */, gps_position_vector_datatype /* oldtype */,
                    &gps_position_array_datatype /* newtype */);

/* fill array of gps_position on sending process with payload */

if (rank == 0) /* sending process */
    MPI_Send(&pos[0], 1 /* count */, gps_position_datatype, 1 /* dest */,
            0 /* tag */, MPI_COMM_WORLD)

if (rank == 1) /* receiving process */
    MPI_Recv(&pos[0], 1 /* count */, gps_position_datatype, 0 /* source */,
            0 /* tag */, MPI_COMM_WORLD, status)

MPI_Type_free(gps_position_datatype);
MPI_Type_free(gps_position_vector_datatype);

```

The example illustrates a first solution to the problem of transferring objects from one process to another. It uses the user-defined datatype techniques as proposed by the MPI standard.

Unfortunately it contains redundant code. The declaration of the members which must be transferred obviously cannot be avoided. But the type of the members must be explicitly provided even though this information is present in the class definition. Declaring `MPI_FLOAT`, `MPI_INT`

instead of `MPLINT`, `MPLFLOAT` will lead to errors which will stay undetected until runtime and will cause memory corruption. These issues make user-defined `MPLDatatypes` hard to read and maintain, error-prone, and not much fun to use. Probably these are the primary reasons why `MPLDatatypes` are rarely used in practice.

The Boost MPI with skeleton and content

The standard solution allows an efficient transmission of objects to a remote process without unnecessary copies into temporary buffers. Also a repeated transferring of the data structure's content without deleting and reallocating the objects is possible.

To make the solution easier to use the Boost MPI provides an important abstraction: it separates the data structure into *skeleton* and *content*.

The *skeleton* describes the structure of a source object by providing information about the types, pointers and sizes of arrays of its members. It can be obtained from any object that is serializable. Once transferred to a remote process, the skeleton can be used to create an object that can accommodate the source object's content by allocating its pointer members and the resizing its array members. The skeleton contains all information in the *Typesig*.

The *content* contains the values of all non-pointer data-members, the payload. It is the data that gets filled into the object. New content can be efficiently filled into the same object provided that the shape of the object, and therefore its skeleton, doesn't change.

To prepare the `gps_positions` example for Boost MPI's solution, we add some code to the class.

```
class gps_position
{
...
private:
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        ar & degrees & minutes & seconds;
    }
...
}
```

Here we specify which members must be transferred by making the class serializable. Although the example only contains integers and a float, more complex types such as arrays of STL containers are handled without additional code.

Note that the type of the members does not need to be specified and that making the class serializable has the side effect that it can also be serialized to a XML stream or any other archive provided by `Boost.Serialization`, see Ramey (2002).

The following code fragment illustrates the sending and receiving of a vector of `gps_positions` with the help of Boost MPI.

```

/* Initialize MPI and the communicator comm */

if (comm.rank() == 0) /* sending process */
{
    std::vector<gps_position> original_vector(4);

    /* fill original_vector with gps_positions */

    // I - send the skeleton containing the size information
    comm.send(1 /* target */, 1 /* count */, skeleton(original_vector));

    // II - send the content containing the values of the gps position
    comm.send(1 /* target */, 1 /* count */, get_content(original_vector));
}

if (comm.rank() == 1) /* receiving process */
{
    std::vector<gps_position> transferred_vector;

    // I - receive the skeleton and resize the datastructure
    comm.recv(0 /* source */, 1 /* count */, skeleton(transferred_vector));

    // II - receive the content containing the values
    comm.recv(0 /* source */, 1 /* count */, get_content(transferred_vector));
}

```

Even though the example is more general than the previous one (the size of the vector get transmitted and must not be known globally), the code is much smaller and clearer.

In step I the *skeleton* gets created and transferred to the receiving process, where the data structure gets prepared. In the example `transferred_vector` gets resized to hold four `gps_positions`.

The content, being the values of all non-pointer members, is transferred in step II. With the call to `get_content(...)` the `MPLDatatype` is created from `original_vector` on the sending and from `transferred_vector` in the receiving process. Since the *skeleton* ensures that the structure of the objects match, the implied *Typesig* is the same and the values of the data-members are transferred correctly.

The transmission of the content can be repeated without destroying and reallocating the objects. The new content gets filled into the old structures and overwrites the previous values. Of course this only works as long as the structure of the object does not change. If the structure of the object is altered, a retransmission of the skeleton becomes necessary.

The Boost MPI's Internals

Internally the Boost MPI works in close conjunction with the Boost Serialization library. For the creation of the *skeleton* as well as for the *content* Archivers (as defined in the Boost Serialization library) are at work.

The Archiver `packed_skeleton_oarchive` creates archives containing the *skeleton* information, i.e. the structure, by filtering out data members and serializing only type, pointer and the array size information of an object. The resulting binary representation is stored in a buffer and is typically sent to the receiving process with the help of the `packed_[i|o]archive`.

The Archiver `content_oarchive` ignores the skeleton information. To create the `MPI_Datatype` from the content, i.e. from all non-pointer members, it uses `mpi_data_type_primitive`.

Conclusion

The paper shows the Boost MPI's *skeleton* and *content* concept, which provides the abstraction of separating structure from content to easily optimize parallel applications which need to transmit the same object many times across process boundaries. Internally the implementation is based on `MPLDATATYPES`, which is opaque to the user. This makes their application less error-prone and eventually used more widely.

References

- Gregor, D. and Troyer, M.: 2005, *Boost.Mpi Library*, http://www.boost.org/doc/libs/1_40_0/doc/html/mpi.html.
- MPI-Forum: 1996, *MPI: A Message-Passing Interface Standard*, MPI forum, <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>.
- Ramey, R.: 2002, *Boost.Serialization Library*, <http://www.boost.org/libs/serialization/doc/>.
- Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J.: 1996, *MPI: The Complete Reference*, The MIT Press, <http://www.parallelsystems.ch/ressources/mpi-book>.